## Helmut Igelbach, Michael B. Zirple

# PC-FORTH©

## A FORTH interpreter/compiler for the PC-1500 with additional functions for controlling the MC-12 system

# Table of contents

**1. Commissioning** .......... 1-1

1.1 Installation .......... 1-1
1.2 Cancellation and recall .......... 1-2
1.3 Input and output .......... 1-3

**2. Introduction** .......... 2-1

2.1 The stack .......... 2-1
2.2 Variables and constants .......... 2-4
2.3 Word definitions .......... 2-6
2.4 Editing Programs .......... 2-8
2.5 Control structures .......... 2-10
2.6 I/O Commands .......... 2-13
2.7 PC-FORTH and BASIC .......... 2-16
2.8 Language extensions .......... 2-17
2.9 Vocabularies .......... 2-22

**3. The PC-FORTH Vocabulary** .......... 3-1

**4. Examples: Useful Definitions** .......... 4-1

4.1 Printer Control .......... 4-1
4.2 Miscellaneous .......... 4-2
4.3 String variables .......... 4-3

**Appendix**

A. Error messages .......... A-1
B. Storage areas and cold start values .......... B-1
C. Dictionary, compiler, interpreter .......... C-1
D. FORTH words for the MC-12 system .......... D-1

# Foreword

The Forth language is revolutionary in many ways. As a relatively young language — it was developed in the USA at the beginning of the 70's — it has only recently conquered a wider audience in Germany. Recently, you will find articles about Forth in almost all journals.

FORTH is now available for most computers. The Forth implementations range from home computers to almost all personal computers to minicomputers. PC-FORTH is the first full-fledged Forth implementation for the PC-1500, which combines both the speed and the scope of performance of a standard Forth. (Only the floppy disk management via SCREEN and BLOCK commands is not implemented.)

FORTH programs are portable. Thanks to the Forth Interest Group (FIG), the FORTH core is well standardized. PC-FORTH complies with the FIG standard, so you can transfer most programs from journals or books directly.

FORTH is fast. Depending on the application, PC-FORTH is 5 to more than 30 times as fast as the PC-1500 BASIC. With PC-FORTH, the PC-1500 is even faster than an Apple II or IBM PC, in BASIC, in many applications and thus reaches a new performance class.

FORTH requires very little disk space. Most compilers inflate the program length enormously, so they are hardly suitable for computers with less than 32K RAM. Due to Forth's almost unbelievable code efficiency, the PC-1500 also reaches a new class here.

FORTH is structured. As in other modern high-level languages (PASCAL, C), Forth contains nestable structures for branching and looping.

FORTH is an interpreted language. Therefore, programs can be developed interactively just as easily as with a BASIC interpreter.

FORTH is a compiled language. Programs are translated by Forth into a very fast intermediate code, so that the execution speed is close to some conventional compilers.

FORTH is close to the machine. Therefore, in many cases, Forth can be used as a replacement for assembler programming.

FORTH can be extended in any direction and thus optimized as a problem-oriented language for the respective field of application.

FORTH is an open language. Since Forth does not impose straitjackets on the programmer, almost any data and program structures can be realized.

However, Forth programming requires a thorough rethink. In the beginning, Forth may seem difficult to some, especially if he has already become accustomed to other programming languages. However, once you are familiar with the new structures, it quickly becomes apparent how simple and powerful this language is.

In addition to the command descriptions, this guide contains only a brief introduction. If that's not enough, we recommend our upcoming FORTH introductory book.

Forth opens up a new dimension of programming. You too will be amazed by the possibilities of this language if you have been familiar with Forth for a while.

Michael B. Zirpel

Gut Wildschwaige, October 1984

## Chapter 1

# Commissioning

The PC-FORTH system consists of two parts: the Forth code and a BASIC part, which is responsible for the input and output.

On the one hand, this ensures that Forth programs run at optimum speed between input and output, and on the other hand, the input and output can be easily changed in BASIC.

MC-12 FORTH runs on any MC-12 (A) system, equipped with a PC-1500 or PC-1500A and a CE-161 memory expansion module.

## 1.1 Installation

First assemble your MC-12 system and don't forget to insert a CE-161 into the PC-1500.

- Switch the PC-1500 to PRO mode and enter the command:

  'NEW 0'

- Enter the command 'RVSLOAD'
- Switch the PC-1500 to RUN mode
- Type 'RUN' to start FORTH

The computer reports with the display

```
PC—FORTH V1.1
```

The MC-12 FORTH is now installed. To try this, type the following (the previous message will now be overwritten):

```
. CPU
```

where you complete the entry with ENTER. The following will appear on the display:

```
FH5801 OK
```

This is the type designation of the CPU (Central Processing Unit) in the PC-1500 built-in microprocessor, followed by the obligatory 'OK' with which Forth confirms the execution of a command.

The BASIC start pointer is &3800. If you have created a FORTH program, save the BASIC part as usual. When reloading your program, first enter the command 'RVSLOAD' to initialize the FORTH. You can overwrite the standard basic part by reloading your program.

## 1.2 Cancellation and recall

Because all inputs and outputs of the Forth system run via a BASIC program, you can press the BREAK key instead of an input at any time and interrupt the work with PC-FORTH exactly as you are used to from BASIC programs.

There are three ways to recall Forth:

**Warm start (DEF F)**

With DEF F, Forth is restarted if you have previously left it with BREAK. With DEF F you will normally call PC-FORTH. (However, the stack of Forth is deleted, see below)

### Cold start (DEF C or RUN)

With DEF C or RUN, Forth is started after it has been loaded from the cassette.

Otherwise, the cold start serves to initialize Forth, and can also be used to "clean up": after the cold start, the Forth system is back to the state in which it was loaded from the cartridge. In particular, all words added by the user will be deleted.

### Hot start (DEF H or GOTO"H")

The hot start is a special function from PC-FORTH and serves to resume the interrupted FORTH program at the point of interruption after a jump into the BASIC.

If the DEF H command is issued manually while Forth is waiting for an input, the last input is automatically repeated.

## 1.3 Input and Output

The input and output of PC-FORTH runs via a BASIC program. This allows you to work with the PC FORTH system, as you are used to from BASIC.

### Input

The input of commands with PC-FORTH is similar to what you are used to from BASIC programs, since all inputs run via an INPUT command in the BASIC part of the Forth system.

However, the input is designed in such a way that the last message of the FORTH system remains in the display and is overwritten by the new input. By pressing the ENTER key without input, however, the display can be deleted at any time.

## Output via display

The display of the PC-1500 is only 26 characters. Therefore, special measures have been taken for the display of longer lines.

If several or longer lines are output via the display, Forth stops outputting each time the display is filled. The output can be continued by pressing the ENTER key. The character ' ~' at the beginning of the line indicates that PC FORTH is not yet ready for input but continues an output.

Example:

Start PC-FORTH with DEF F
Enter the command VLIST

On the display, all commands from PC FORTH are now displayed. Each time you press the ENTER key, the next line is displayed. Use the • key to cancel VLIST.

You can use the I and t keys to switch the TRACE on and off during the output stop (compare the commands TRC, TRACE and TROFF).

By displaying the cursor '__' on the first display position, PC-FORTH shows that it is ready for input.

## Output via CE-150

All inputs and outputs of PC-FORTH can be logged on the CE-150 printer. DEF L allows you to turn the printer on and off.

Enter the command VLIST once with the printer switched on, you will then receive a listing of the vocabulary.

If the printer is switched on, PC-FORTH no longer pauses the output, but continues to work continuously.

However, if you press any key on longer print sequences such as VLIST (you may have to hold down the key for a while), PC-FORTH stops the output. You can then activate the TRACE, resume the expression or cancel the expression with • in VLIST.

## Output via CE-158

Instead of the CE-150 printer, you can use other printers for output via the CE-158 interface. To do this, interrupt PC-FORTH with BREAK and enter the corresponding BASIC commands that are necessary for LPRINT output to your device (SETDEV PO).

Then the external printer is controlled in the same way as the CE-150 and can be switched on and off with DEF L.

## Chapter 2

# Introduction

This section covers the basic characteristics of Forth in a crash course. Of course, only the most important points can be illuminated. A detailed description of all PC-FORTH commands can be found in Chapter 3.

At first glance, Forth may seem too low level and close to the machine to some high-level language programmers.

But Forth is designed for extension: You can develop the language in any direction as complex as you like in its compiler properties as well as in the running time functions. Almost any data types, language structures and user functions can be implemented in Forth.

After a period of getting used to it, you will quickly realize that Forth is a much more powerful programming language than BASIC and surpasses other languages such as C in some respects.

Forth, however, requires a certain rethinking of traditional programming languages, as other basic concepts are used: the stack and the postfix notation, the mixture of an interpreter and a compiler, which is itself programmed in Forth and can be extended as well as the runtime vocabulary, the dictionary concept and the different vocabularies are new and unusual for most high-level language programmers as is the closeness to the machine level.

## 2.1 The stack

The programming language Forth, as is known from some calculators, uses the so-called Reverse Polish Notation (RPN), sometimes referred to as postfix notation.

This means: If an operation is to be performed with any data, you first enter the data, then the operation.

Example:

It should be calculated 123 + 5 - 17.
In the RPN one writes:
123 5 + (123 + 5 = 128)
17- (128 – 17 = 111)

In order for the whole thing to work properly, Forth works with a so-called stack, in German called 'Stapel' (Stack) or ‘Keller’ (basement).

If numbers are entered, they are placed on the number stack. The last number entered is then at the top of the stack, the penultimate number entered at the second top, and so on.

With the command ‘.S’ you can see what's on the stack, the bottom number appears on the left, the top number on the right.

Example:

| Input: | Display: | Note: |
|---|---|---|
| DEFF | PC-FORTH 1.1 | Warm boot of PC-FORTH |
| 11 22 33 | OK | Now the entered numbers are on the stack |
| .S | 11 22 33 OK | 33 is above, 11 at the bottom |

When an operation is performed, the operation retrieves the required data from the stack and places the results back on the stack.

The data is always taken from the top of the stack. If there are other values on the stack, they remain unchanged.

When added with ‘+’, the top two numbers are taken from the stack and added. The result of the addition is then placed on the stack.

Example:

| Input: | Display: | Note: |
|---|---|---|
| .S | 11 22 33 OK | still on the stack |
| + .S | 11 55 OK | (33 + 22 = 22) |
| + .S | 66 OK | (55 + 11 = 66) |

At first, you may find the postfix notation a bit difficult, because with longer intricate sequences it requires a certain rethinking. After a certain period of acclimatization, however, it should no longer cause you any major difficulties.

All operations in Forth are usually described by specifying what data they expect on the stack and what data they leave on the stack. For example, the + operation is described as follows:

+ ( n1 n2--n3 ) n3 = n1 + n2

This means: + expects two numbers on the stack, n1 and n2, which are removed from the stack and replaced by n3, the sum of n1 and n2.

With this form of description, you can always see at a glance what happens to the stack.

Here are short descriptions of the most important arithmetic operations in Forth:

| | | |
|---|---|---|
| + | ( n1 n2 - n3 ) | n3 = n1 + n2 |
| - | ( n1 n2 - n3 ) | n3 = n1 - n2 |
| * | ( n1 n2 -- n3 ) | n3 = n1 * n2 |
| / | ( n1 n2 -- n3 ) | n3 = n1 / n2 |
| MOD | ( n1 n2 - n3 ) | n3 = Division remainder of n1 / n2 |
| ABS | ( n1 -- n2 ) | n2 = absolute value of n1 |
| MINUS | ( n1 - n2 ) | n2 = - n1 |

Forth works with integers. Most operations are performed with signed 16-bit numbers, the value range is then -32768... +32767.

If you want to do longer calculations in Forth, you have to think a little ahead, but you get used to it quickly.

A series of commands for batch manipulation makes the work easier.

| | | |
|---|---|---|
| DROP | ( n~ ) | deletes top value |
| DUP | ( n - n n ) | duplicates the top value |
| SWAP | ( n1 n2 - n2 n1 ) | swaps the top two values |
| OVER | ( n1 n2 – n1 n2 n1 ) | copies second value upwards |
| PICK | ( ... n1 -- ...n2) | copies the n1th value upwards<br>(counting from n1 with 0) |

The command is used to output a number from the stack. (i.e. just one point). Note that the output value is deleted from the batch at the same time.

| | | |
|---|---|---|
| | (n~ ) | Output of the top value |

The use of the stack offers several advantages: you do not need variables to store intermediate results, but you can leave the values on the stack. The transfer of values to your own subroutines or functions takes place uniformly via the stack, just as with the commands of the Forth language.

Forth uses another stack for various (internal) purposes, the so-called return stack (R stack). To differentiate, the normal stack is sometimes referred to as the C stack (computation stack).

## 2.2 Variables and constants

Unlike BASIC, variables must be defined before use. The word VARIABLE is used for this purpose, which is used in the following form:

n VARIABLE Name

N specifies the initial value of the variable, which is stored in the Forth dictionary under the specified name. The name can be up to 31 characters long.

Example:

| Input: | Display: | Note: |
|---|---|---|
| 0 VARIABLE X | OK | X with initial value X = 0 |
| 5 VARIABLE Y1A | OK | Y1A with initial value Y1A = 5 |

If you want to get the contents of a variable on the stack, this is done by calling the variable name followed by © (with spaces in between).

If a value is to be stored in a variable, this is done by name call followed by ! (spaces in between).

Example:

| Input: | Display: | Note: |
|---|---|---|
| Y1A @ .S | 5 OK | Contents of Y1A on the stack |
| 7 + 3 * .S | 36 OK | |
| X ! | OK | Save to X |

The variable names are entered into the Forth dictionary during definition, while at the same time memory is reserved for the value of the variable. If a variable name is called, this call leaves the memory address on the stack where the value is stored. The commands @ and ! then work with the memory addresses:

| | |
|---|---|
| ( a - n ) | loads the number n stored from address a |
| ( na- ) | stores the value n from address a |

Like variables, constants can be defined, the value of which is stacked by simply mentioning the name. This is done with the statement:

n CONSTANT Name

that enters a constant with the value n under the specified name in the dictionary. When the name is called, the value n is placed on the stack.

## 2.3 Word definitions

There is no distinction in Forth between commands, functions, operators, procedures, or subroutines. In Forth, there are only words that are used largely equally.

The existing words such as + or DUP form the basic vocabulary from which the user forms new words, which are included in the Forth dictionary on an equal footing with the basic vocabulary.

It is part of Forth's 'philosophy' that programs consist simply of defining new Forth words. Each program automatically has a name and is a new command of the Forth language.

The word names can be formed from any characters, only the space separates individual words from each other. The maximum name length is 31 characters, all of which are taken into account when distinguishing names.

A word definition has the following form:

```
: Name Wordsequence ;
```

This defines the new word 'Name' and enters it in the dictionary. The word sequence specified in the definition is compiled and stored as Forth pseudocode in the dictionary immediately after the name.

If the name is called later, the word sequence specified in the definition is executed.

It is common Forth practice to shorten elementary command sequences with new words for code optimization if they are used more frequently.

Example:

Definition of the word QUAD, which squares the top stack value. Input (do not forget space, at the end of the input ENTER):

```
: QUAD DUP * ;
```

Now QUAD is defined and can be used:
3 QUAD . now returns the result 9

However, QUAD can also be used in new definitions, e.g. in the definition of the word CUBE, which exponentiates the top stack value with 3:

```
: CUBE DUP QUAD * ;
```

3 CUBE . returns the result 27

In Forth, you can add new names to words of the basic vocabulary at any time, e.g. to use abbreviations or to Germanize Forth.

Example:

```
:PRINT. ;      ( PRINT instead of .)
: D DUP ;      ( D instead of DUP )
```

You can also define new words under the name of already defined words. Then the message "~ new" appears, but the definition is accepted.

Example:

```
: . DUP. ;     ( non-deleting number output )
```

Forth always takes the last definition of a name as the current definition.

It is part of Forth's 'philosophy' to keep word definitions as simple and short as possible.

If a longer program is to be realized, this program will not be packed into a single word definition, but the program will be broken down into meaningful parts as far as possible. A word is defined for each of these parts. Each of these words is first tested before defining further words. Finally, a 'noun' can be defined that summarizes the 'sub-words'.

Forth offers in a simple and elegant way the possibility to realize your own command languages in more complex program systems: You define corresponding words for the individual operations, which can be called just like other words. This saves the programming of a main program that calls individual operations via menus or commands. The Forth system does this all by itself and at the same time offers the possibility to program in the new command language.

## 2.4 Editing the programs

In the Forth interpreter mode, you can define new words at any time, as has been done in the previous examples. Since the corresponding programs are compiled in the same way, it is no longer possible to edit them, e.g. to make an insertion. The only editing option in the Forth interpreter mode is the command FORGET, which is available in the form:

```
FORGET Name
```

is used. It causes all definitions that have been made after the latest definition of the specified name (including name) to be deleted.

You can then type in the changed definitions again.

Because this is of course not sufficient for longer programs, Forth offers the possibility to take program texts from an editor and compile them.

The FIG-Forth standard uses so-called SCREENs for this purpose. These are text memories on the floppy disk, which always contain 16 lines of 64 characters and can be written to with the SCREEN editor. These SCREENs are numbered consecutively and can be specifically addressed by the Forth interpreter.

PC-FORTH takes a different approach by using the BASIC program memory and the BASIC editor instead of the SCREENs. This gives PC-FORTH users the advantage of a more comfortable editor and also furthers possibilities to mix BASIC and Forth (see below).

To enter Forth source programs, switch (after exiting the Forth interpreter with BREAK) to the PRO mode of the PC-1500 and write the Forth commands into normal BASIC program lines, where quotation marks are entered after the line number.

Example:

```
200": QUAD ( n -- n*n ) DUP * ;
210": CUBE ( n--n*n*n ) DUP QUAD * ;
```

As you can see in the example, the parentheses in Forth are used to include comments in the program. The opening parenthesis must be followed by a space, as it is a Forth word. Forth ignores anything after an opening parenthesis until the next closing parenthesis or end of line.

However, when entering Forth source programs, you must not change or delete the BASIC lines belonging to the Forth system. These lines have numbers smaller than 200.

The entered Forth source program is followed by a line with the word P> , which marks the end of the Forth source code.

Example:

```
220" P >
```

To compile the source code, start PC-FORTH and then enter the line number where the Forth source code begins, and then run the command <P .

Example:

```
200 < P
```

This command simply redirects the Forth input. Instead of the keyboard, the input into the Forth is made from the BASIC program memory. The P> command then switches back to keyboard input.

This reading from the BASIC program memory takes some time. When it is completed, any messages and then the obligatory OK are issued.

Forth source programs can be edited or saved on cassette like ordinary BASIC programs. You should not be bothered by the BASIC lines required for the Forth system and save them on the cassette.

## 2.5 Control structures

Forth supports structured programming through nestable structures for looping and branching, just as is common in other modern programming languages.

However, these structures may only be used within word definitions (programs).

Conditions are also specified in the postfix notation for the structure words: first comes the condition, then the respective control command (e.g. IF).

To formulate the conditions, a number of comparison commands are available, which leave a 0 or 1 on the stack, depending on the outcome of the comparison.

The most important comparison commands are:

| | | |
|---|---|---|
| < | ( n1 n2 -- n3 ) | n3 = 1 if n1<n2, otherwise n3 = 0 |
| = | ( n1 n2 -- n3 ) | n3 = 1 if n1 = n2, otherwise n3 = 0 |
| > | ( n1 n2 -- n3 ) | n3 = 1 if n1>n2, otherwise n3 =0 |
| 0< | (n1--n2 ) | n2 = 1 if n1<0, otherwise n2 = 0 |
| 0= | (nl-n2 ) | n2=1 if n1=0, otherwise n2 = 0 |

The command 0= is often used to invert a condition and is therefore also referred to as NOT in some Forth dialects. (If you want to do the same, you can start with: NOT 0= ; place the NOT command in PC-FORTH. The remaining comparisons such as >= can also be realized in a simple way : >= < NOT ; etc.)

Example:

```
X @ 3 >                 ( Test whether the contents of variable X @ > 3 )
```

Conditions can be linked to the available logical commands.

| | | |
|---|---|---|
| AND | ( n1 n2 -- n3 ) | n3 = n1 AND n2 (Bitwise And) |
| OR | ( n1 n2 -- n3 ) | n3 = n1 OR n2 (Bitwise Or) |
| XOR | ( n1 n2 -- n3 ) | n3 = n1 XOR n2 (Bitwise Exclusive-Or) |

Example:

```
X @ 3 > X @ 10 < AND            (Test whether X>3 and X<10)
```

The control structures cause branches in the program as the program runs, depending on whether conditions are met or not.

The conditions are always taken from the stack: the value 0 indicates that the condition is not met, any other value indicates that the condition is met.

The following tree commands exist in Forth:

```
Condition IF Wordsequence ENDIF
```

The word order between IF and ENDIF is only traversed if the value taken from the stack at IF is not zero.

```
Condition IF Wordsequence ELSE Word order2 ENDIF
```

The word sequence is iterated if the value taken from the stack at IF is not zero, otherwise the word sequence2. In both cases, the program will continue after ENDIF.

```
BEGIN Wordsequence condition UNTIL
```

The word order between BEGIN and UNTIL is traversed. If the value taken from the stack at UNTIL is zero, the word order is run again, otherwise the program is continued after UNTIL .

BEGIN WordSequence Condition WHILE WordSequence2 REPEAT

Word sequence is traversed. If the value removed from the stack at WHILE is not zero, the word sequence continues2 and then repeats word sequence. If the value taken from the stack is zero for WHILE, the program run continues after REPEAT.

A counting loop structure (similar to FOR... NEXT) is also available in Forth. These are the words DO and LOOP or LOOP, which are used as follows:

n1 n2 DO WordSequence LOOP

DO takes the values n1 and n2 from the stack. The loop counter is set to the initial value n2, then the word sequence is iterated. In LOOP, the loop counter is increased by 1 and compared with the barrier n1. If the limit is reached or exceeded, the program is continued behind LOOP, otherwise the word sequence between DO and LOOP is repeated.

If you do not want to count in one-step increments, use the + LOOP command instead of LOOP as follows:

n1 n2 DO WordSequence n3 +LOOP

+LOOP takes the value n3 from the stack and adds it to the loop counter, otherwise the whole thing works like DO and LOOP.

Example:

: TEST 10001 1 DO LOOP ;

This empty loop is traversed l0000 times and can serve as a benchmark test. If you call the word TEST, it takes about 4 seconds to execute (compared to about 140 seconds for FOR-NEXT in BASIC, PC-FORTH is 35 times faster.)

All loop and branch structures can be nested within each other, according to the rules of structured programming.

## 2.6 Input/Output Commands

The input and output of numbers can be done in Forth in any number system. With the commands HEX and DEC you can switch to the hexadecimal system or switch back to the decimal system.

Example:

| Input: | Display: | Note: |
|---|---|---|
| HEX | OK | Hexadecimal system |
| 14 8 + . | IC OK | Result hexadecimal |
| FF DEC . | 255 OK | Decimal again |

Other number systems can be selected by storing the value of the number system base in the system variable BASE. 16 BASE ! is e.g. identical to HEX, with 2 BASE ! you can switch to the binary system.

Forth first tries to interpret each input as a word. Therefore, it may be necessary to mark e.g. hex numbers with leading zeros to avoid confusion with words. For example, the hex number DEC must be entered as 0DEC, because DEC is considered a command.

The following commands are available for number output:

| | | |
|---|---|---|
| | ( n — ) | Standard output of a number |
| .R | ( n1 n2 — ) | prints n1 right-aligned in a cell of length n2 |

In addition to the signed 16-bit numbers used so far, Forth also knows unsigned 16-bit numbers. The value range is then 0...65535, as an abbreviation for such numbers a (address) or u (unsigned) is used.

You can output unsigned numbers with the following command:

| | | |
|---|---|---|
| U. | ( a - ) | Output of an unsigned number |

In addition, Forth contains the double-length (double-precise) 32-bit numbers, where the value range is -2147483648... +2147483647.These numbers are abbreviated with d.

Such double-length numbers are entered by using a decimal point when entering numbers. They are issued with the commands:

| | | |
|---|---|---|
| D. | ( d - ) | Output of a double-length number d |
| D.R | ( d n - ) | Output of d right-aligned in the field of length n |

Example:

| Input: | Display: | Note: |
|---|---|---|
| 1234567. | OK | Entering a double number |
| D. | 1234567 | Output of a double number |

These double-length numbers occupy two stack positions (4 bytes versus 2 bytes) compared to simple numbers. According to the Forth convention, the higher-order half is at the top of the stack.

You will find in the vocabulary in chapter 3 a whole series of commands for handling the double numbers, e.g. 2DUP, D +, DABS. However, some words are missing compared to simple numbers and must be defined by the user if needed.

The position of the decimal point entered with double numbers, which may be anywhere within the entered number, can be taken by the programmer from the system variable DPL, so that the programming of fixed-point numbers is possible.

Texts are output with the command ", the end of the text is marked with ". The text output is therefore in the form:

```
." text to be output"
```

(PC-FORTH uses " instead of always" for technical reasons.)

Example:

| Input: | Display: | Note: |
|---|---|---|
| : TEST1 ." PROBETEXT " ; | OK | |
| TEST1 | PROBETEXT OK | output followed by OK |

All outputs from Forth first come into a buffer. Only when it is full or when the CR command is issued, the buffer is actually output.

| | | |
|---|---|---|
| CR | ( — ) | Output of buffer and start of a new line |

Example:

| Input: | Display: | Note: |
|---|---|---|
| : TEST2 11 . CR | | Test Definition |
| 22 . 33 . CR ; | OK | |
| TEST2 | 11 | Continuous output, each new line |
| | 22 33 | when printer on, otherwise |
| | OK | output stop after each line |

The following additional commands are available for output:

| | | |
|---|---|---|
| EMIT | ( n --) | prints the character with the ASCII code n |
| TYPE | ( an- ) | prints n characters stored from address a |
| SPACE | ( - ) | returns a space |
| SPACES | ( n --) | outputs n spaces |

The following commands are available in PC-FORTH for programmatic input:

| | | |
|---|---|---|
| EXPECT | (an- ) | Enter a maximum of n characters, which are stored from address a followed by a 0. |
| KEY | ( - n ) | Entering a character, returns its ASCII code n |

The function NUMBER is available for converting entered characters into a number (corresponds to the VAL from BASIC):

| | | |
|---|---|---|
| NUMBER | ( a ~ d ) | converts the string stored from address a to a double number. |

Of course, the strings entered with EXPECT must be stored somewhere. Forth provides a memory area above the dictionary for this purpose, but its position shifts after each entry in the dictionary. The PAD command returns the current start address of this storage area.

To store individual characters, Forth can also be used to address individual memory cells or memory areas:

| | | |
|---|---|---|
| C@ | ( a - n ) | loads contents n of the memory cell a onto the stack |
| C! | (na- ) | stores n (n = 0...255) at address a. |
| ERASE | (an-) | deletes n bytes from address a (with zeros) |
| BLANKS | (an-) | fills n bytes from address a with spaces |
| FILL | ( a n1 n2 - ) | fills n1 bytes a with the value n2 (n2 = 0...255) |
| CMOVE | ( a1 a2 n - ) | copies n bytes from address a1 to a2 |

If texts are stored, the string is preceded by a length byte containing the number of stored characters or one or more zeros are appended to the string as final characters.

## 2.7 PC-FORTH and BASIC

PC-FORTH is designed to work with the PC-1500 BASIC, as shown by the introduction and use of the BASIC Editor.

You can start a BASIC program from a Forth program. This is done by the BAL command:

| | | |
|---|---|---|
| BAL | ( n - ) | jumps to BASIC line with number n (n=1...255) |

With this command, only lines with a number from the range 1...255 can be called. However, the range can be easily extended via an intermediate jump.

The return to the FORTH takes place with the BASIC instruction GOTO"H", the interrupted Forth program is then continued after the interruption point.

In BASIC however, the variables O, S, L, I$(0) and O$(0) must not be changed, as they are required by FORTH.

You can output BASIC lines of text in a Forth program. (BASIC lines of text begin with quotation marks and do not contain BASIC commands).

| | | |
|---|---|---|
| . LINE | ( n - ) | prints the BASIC line of text with the number n. |

PC-FORTH uses this command when issuing the error messages stored on BASIC lines.

For manipulation of BASIC lines (e.g. RENUMBER), the B# command returns the memory address at which the text of a BASIC line begins.

Data can be taken from BASIC variables with the @: command and stored in BASIC variables with the !: @ command.

These commands are displayed in the form:

@ : "Variablename" or !: "Variablename"

The variable name must be enclosed in quotation marks. However, these commands can only be read via BASIC lines and not directly from the keyboard. The exact description can be found in Chapter 3.

## 2.8 Language extensions

Actually, every newly defined word is an extension of the Forth language: you can type it directly and have it executed, but you can also use it within another definition, compiling the call to the new word.

However, if you want to extend the Forth compiler, which translates the word sequences given in the definitions, with new words or define new variable types, there are various things to consider, which will now be briefly examined. (See Appendix C for more detailed explanations of how the compiler works and the structure of the dictionary.)

In the case of a definition, the name of the newly defined word and some identifiers are first entered in the dictionary. This is followed by the so-called parameter field, into which the word sequence specified in the definition is compiled and stored.

A number of commands are available for managing the dictionary. To write individual bytes into the dictionary, Forth uses the following words, among others:

| | | |
|---|---|---|
| HERE | ( - a ) | returns the address of the first free byte after the previous entries |
| , | ( n -- ) | adds n (2 bytes) to the dictionary starting from HERE and increments HERE by 2 |
| C, | (n - ) | adds n (1 byte) to HERE and increments HERE by 1 |
| ALLOT | (n - ) | leaves n bytes free in dictionary (increments HERE by n) |

You need these commands, for example, if you want to create variables of a new data type (e.g. fields or text variables) and use storage space in the dictionary for this.

To define new data types, the words <BUILDS and DOES> are used in the following form:

```
: Typename <BUILDS Wordsequence1> Wordsequence2 ;
```

The specified Typename is thus made into a definition word (such as VARIABLE or CONSTANT).

If you want to define an object of the new type, this is done in the form:

```
Typename Objectname
```

The word sequence between <BUILDS and DOES> is executed from the type definition.

If you want to call an object defined in this way, this is done as usual by specifying the object name:

Objectname

The word sequence is executed, which is after DOES> in the Typedefinition, where the address of the parameter field (PFA) of the object is passed on the stack.

Example:

To store double-long numbers, the commands 2! and 2@ are available, but no double variables, the definition of which can be made up:

```
: 2VARIABLE         (defines the type double variable)
<BUILDS
HERE 2!             (takes initialization value and saves it from HERE)
4 ALLOT             ( the value of the variable occupies 4 bytes in the
dictionary)
DOES> ;             (leaves parameter field address, does nothing else)
```

Application of 2VARIABLE:

```
0. 2VARIABLE A      ( defines the variable A with the initial value 0)
12345678. A 2!      ( stores the value 12345678 in A)
A 2@ D.             (outputs the contents of A)
```

The type 2CONSTANT can be defined analogously:

```
: 2CONSTANT         (defines the type double constant)
<BUILDS
```

```
HERE 2!          (enter the value)
4 ALLOT
DOES>
2@ ;             (loads the value on the stack)
```

With <BUILDS and DOES> you can implement the most diverse data types or structures in an elegantly simple way, the associated operations are then defined as words.

However, it is sometimes necessary to influence the compiler, which does not cause any major difficulties in Forth.

Usually, when you enter a definition, all the entered words are compiled and executed later in the call.

Example:

```
: TEST HEX 7750 C@ . ;
```

If this command is executed, Forth switches to the hexadecimal system and outputs the contents of the memory cell 7750. If the decimal number system was set while entering the definition, then 7750 means decimal 7750, that is, hexadecimal 1D4C, and the corresponding numerical value was compiled. TEST then displays the contents of the memory cell hex 1D4C.

However, there is the possibility of executing commands even during a definition. To do this, with (( the compilation is interrupted and with)) is resumed.

Example:

```
: TEST (( HEX )) 7750 C@ . ;
```

Now it is switched to the hexadecimal system during the definition and 7750 is compiled as hex 7750, that is, decimal 30544. During the call of TEST, the set number system is not changed at all.

With the help of (( and )), a calculation can also be performed during a definition, for example to include the result in the program as a constant with the command LITERAL:

Example:

```
: TEST DUP 5 + * ;   ( n -- n*(n + 5)
```

The number 5 is included as a constant in the compiled program; however, it can also be calculated during compilation without changing the compiled program:

```
: TEST DUP (( 23 + )) LITERAL + * ;
```

So if the commands are to be executed during a definition and not compiled, then (( and )) are used.

But there are also words in Forth, which are always executed immediately. They are called immediate words. Such immediate words, even if they are used within a definition, are executed immediately, without (( and )) having to be used for this purpose. These immediate words include, for example, the semicolon ; or ((.

If you want to define your own immediate words, this is done with the error IMMEDIATE, which is given immediately after the definition of a word is completed in order to mark the word as immediate.

However, if you want to compile immediate words such as the semicolon in a definition, you have to forward the word ((COMPILE)).

Example:

```
: DEF((COMPILE)) :;              (: can now be used instead of DEF)
: ENDDEF((COMPILE)) ; ;          (; can be used instead of ENDDEF)
IMMEDIATE                        (ENDDEF must be like; immediate)
```

Another category is the Forth words, which are used only within definitions and are executed immediately, but at the same time cause the compilation of other words.

All structure commands such as IF or ENDIF belong to this category: they are just words. During compilation, you check the structure of the program based on values that you leave on the stack or take from the stack. However, they will not be included in the program, but instead they cause corresponding jump commands to be compiled into the program. To do this, use the COMPILE command.

Example:

```
: HEXA HEX COMPILE HEX ; IMMEDIATE
: TEST HEXA 7750 @ . ;
```

Since HEXA is an Immediate command, it is also used to switch to the hexadecimal system within a definition with HEXA. At the same time, HEXA initiates the compilation of HEX, so that the switchover to the hexadecimal system is also included in the compiled program and is therefore also performed when calling TEST.

## 2.9 Vocabulary

Previously, the defined words were simply entered in the dictionary without specifying in which part of the dictionary they should be included.

In the dictionary, however, you can define different vocabularies and thus combine entire groups of words under one name. At the beginning, however, there is only a single vocabulary in PC-FORTH, which bears the name FORTH and is congruent with the entire dictionary.

The definition of a vocabulary is done in the following way:

```
VOCABULARY Vocabularyname IMMEDIATELY
```

This creates a new vocabulary under the specified name. If the definitions are to be entered in this new vocabulary, this is done by calling the vocabulary name followed by DEFINITIONS:

DEFINITIONS of Vocabularyname

All the following word definitions will now be entered into the called vocabulary.

If definitions are to be re-entered into the basic vocabulary FORTH, one enters:

FORTH DEFINITIONS

The vocabulary in which the respective definitions are entered is called CURRENT vocabulary.

If a vocabulary is to be activated so that the words defined therein are available for the application, this is simply done by calling the Vocabularyname:

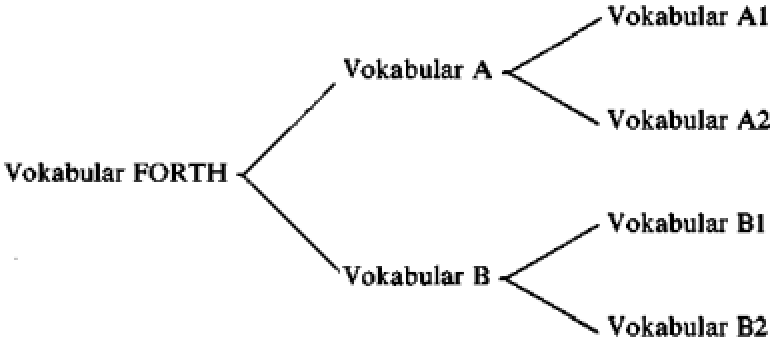
Vocabularyname

All entered words are then searched first in the called vocabulary. If they are found there, they are executed or compiled. If they are not found there, the search continues in the vocabulary in which the called vocabulary was defined. As a result, the words of the FORTH vocabulary are always available.

The currently active vocabulary in which the entered words are searched is called CONTEXT vocabulary.

Depending on the order of definition of vocabularies, tree-like structures can be defined for word search.

Example:



Is generated by:

```
FORTH DEFINITIONS                  (Initial state)
Entry of definitions in FORTH
VOCABULARY A IMMEDIATE             (Definition of vocabulary A)
  A DEFINITIONS                    (A as a CURRENT vocabulary)
  Entry of definitions in A
  VOCABULARY AI IMMEDIATE          (Definition of the vocabulary AI)
    AI DEFINITIONS                 (AI as a CURRENT vocabulary)
    Entry of definitions in AI
  A DEFINITIONS                    (A as a CURRENT vocabulary)
  VOCABULARY A2                    (Definition of vocabulary A2)
    A2 DEFINITIONS                 (A2 as CURRENT vocabulary)
    Entry of definitions in A2
FORTH DEFINITIONS                  (Initial state)
More listings in FORTH
VOCABULARY B IMMEDIATELY           (Definition of vocabulary B)
  B DEFINITIONS                    (B as CURRENT vocabulary)
  Entry of definitions in B
  VOCABULARY Bl IMMEDIATE          (Definition of vocabulary Bl)
     B1 DEFINITIONS                (B as CURRENT vocabulary)
     Entry of definitions in Bl
```

(Initial state)

After:

AA2

the words of the vocabulary A2 can be called.

After:

A A2 DEFINITIONS

further definitions can be included in the vocabulary A2.

Typical application examples in Forth systems for such vocabularies are the EDITOR vocabulary, with which the commands of the SCREEN editor are activated, or the ASSEMBLER vocabulary, with which machine commands can be entered in the mnemonic form.

Vocabularies generally serve to give the defined names a local scope.

Forth treats the entries in the vocabularies as linked lists by entering an address (link address) with each name entry, which points to the last entry belonging to the same vocabulary.

## Chapter 3

# Vocabulary

Below you will find the root vocabulary of PC-FORTH described.

As usual in most FORTH descriptions, the command words are listed in ASCII code order. Since any special characters are also used as command words, an alphabetical order is not sufficient.

Words whose description is in small print are usually used only within FORTH. Most often these are commands that the compiler translates differently than you type them (e.g. instead of IF... ELSE... ENDIF translates various BRANCH commands, of which the user usually does not make use). However, if you want to extend the compiler, it is useful to use these internals.

In the descriptions, deviations from the standard FIG FORTH are expressly pointed out. All commands from FIG-FORTH, which do not concern the virtual memory management of floppy disks or screens, are implemented completely compatible in PC-FORTH with a few exceptions. For most commands, other FIG-FORTH descriptions can also be consulted.

Some deviations from FIG-FORTH concern the naming: [ and ] is basically written as ((and)), and " by" is also replaced. Instead of ' PC-FORTH uses the word TIC.

The command descriptions indicate the effect of the command on the stack.

Thereby marking:

| | |
|---|---|
| n, n1, n2 | signed 16-bit numbers (2 bytes)<br>Value range: -32768 ... +32767 |
| a, a1, a2 | unsigned 16-bit numbers (2 bytes), as they are mainly used for address information Value range: 0... 65535 |

| | |
|---|---|
| d, d1, d2 | signed 32-bit numbers (4 bytes)<br>Value range: -2147483648... +2147483647 |
| ud, ud1, ud2 | unsigned 32-bit numbers (4 bytes)<br>Value range: 0... 4294967295 |

On the left is what must be on the stack before the respective command, on the right is what is found on the stack afterwards.

**!** ( na- )

stores n at the addresses a, a + 1 (the higher-order byte of n comes after a). Will be in the form:

n Variablename !

used to store the value n in a variable.

**!:** ( n- ) or ( an- )

used to pass numerical values or strings to BASIC Variable. Can only be used within definitions that are read in with <P.

Application form:

| | |
|---|---|
| n !: "v" | Transfer of the value n to the variable v |
| an ! : "v$" | Transfer of the string with the length n from Address a to the text variable v$ |

v can also be a BASIC field element instead of a simple variable.

If an error occurs during the evaluation of v (e.g. un-dimensioned field), the error message is ?"basic.

The compiler sets for !: the code address of the associated runtime function (!:) and the expression v or v$ with preceding length byte and following end separator ab. The stored expression can also contain BASIC tokens

Not included in FIG-FORTH.

**!CSP** ( - )
stores the stack level (C-Stack) in the CSP user variable. (Used by the compiler for structural checks.)

**#** ( d1 -- d2 )

puts the last (least significant digit) of d1 into a buffer as a printable ASCII digit. For this, a division is carried out:

d2 = d1/number base (BASE), where the division remainder is converted to an ASCII digit (used within the number output, between < # and #>. )

**# >** ( d - a n )
finished processing a number for the output:
Deletes d, leaves the initial address a and length n of the ASCII digits in such a way that it can be output with TYPE.

**# S** ( d1 -- d2 )
prepares d1 as a printable ASCII numeric sequence.
(Repeats # until d2 = 0.)

**# "** ( n1 ~ n2 )
returns the number of the first occurrence of a character in a character sequence. To be applied in the form:

n1 # "String"

Searches for the character with the ASCII code n1 in the specified character sequence. If the character is found there, then n2 is the number of the first occurrence within the string, otherwise n2 has the value 0 (counting is from left to right from 1). Can be used together with CASE: for control structures.

Instead of ¥, the compiler stores the code address of the associated runtime function (#") and behind it the string preceded by a length byte.

Not in FIG-FORTH.

‘ in (FIG-FORTH)
identical to TIC in PC-FORTH

**&B** ( a1 a2 -- a3 a4 n )

Conversion of (long) hex digit sequences into binary values, can be used to enter machine code:

A sequence of hex digits stored from address a2 in ASCII representation is stored binarily encrypted from address a1. Each 2 digits equals 1 byte. The first character, which is not a hex digit (0 to 9, A to F), aborts processing. However, a space between pairs of digits does not cause a break but is read over. a3 is the address after the last byte of the result; a4 is the address of the character that cancelled processing; n is this sign itself. The length of the string to be converted is limited only by the available disk space. However, it should be even. Otherwise, the last digit is ignored.

Not in FIG-FORTH

**(** ( -- )

initiates comment. The opening parenthesis must be followed by a SPACE; the comment begins behind it. The first closing parenthesis or end of line (ENTER) ends the comment.

**(!:)** ( n -- ) or ( an- )

Runtime function too !: is followed by length byte and BASIC expression, to which an end separator is attached.

**(#~)** ( n1 - n2 )

The #" runtime function is followed by a length byte and a string that is searched for the character with the ASCII code n1. On the stack, n2 is left, where n2 = 0, if the character was not found, otherwise n2 has the number of the first occurrence.

Not in FIG-FORTH

**((** ( " )

interrupts the compile state within a word definition and switches to direct execution. The following words are not compiled, but executed immediately until )) is switched back to the compile state.

In FIG-FORTH [

**((COMPILE))** ( - )

within a word definition, ensures that the next one IN the MEDIATE word is compiled and not executed. IMMEDIATE words such as ; or ((COMPILE)) are usually also executed immediately within definitions, consider IMMEDIATE.

In FIG-FORTH [COMPILE]

**(.")** (FIG-FORTH)

identical (.") in PC-FORTH.

**(.")** ( -- )

Runtime function too ." is followed by length bytes and text. Writes the text to the output buffer.

In FIG-FORTH (.")

**(;CODE)** ( - )

Runtime function ;CODE. Describes the code field of the last defined word with a pointer to the following machine code.

**(@:)** ( -- )

Runtime function for @: is followed by length byte and BASIC expression, to which a colon is appended.

**(+LOOP)** ( n -- )

Runtime function to +LOOP; adds n to the loop counter; if the limit stored by (DO) is exceeded or reached, loop-end treatment follows. Otherwise, jump back behind (DO).

**(ABORT)** (initialized stack)

causes warm start. Executed in the event of an error if WARNING has a value of -1.

**(DO)** ( n1 n2 --)

Writes the initial value n2 and the limit n1 for the loop counter to the R stack (R stack). See LOOP and +LOOP.

**(FIND)** ( a1 a2 - a3 n1 n2 ) or ( a1 a2 -- n3 )

look for a name in the dictionary.

a1 = Name field address of the name to be searched in memory;
a2 = Name field address of the name in the dictionary with which the search should begin;
a3 = Name field address of the found name in the dictionary;
n1 = length byte of the found name (including immediate bit);
n2 = 1 (means: "Name found")
n3 = 0 (means: "Name not found")

**(LINE)** ( n1 - a n2 )
provides a BASIC (text) line for output (as a message).

n1 = BASIC line number
a = your starting address (after the length byte)
n2 = your length byte

After (LINE), the line can be written to the output buffer using TYPE.

In FIG-FORTH, with a different meaning.

**(LOOP)** ( - )
Runtime function to LOOP increments the loop count on the R stack by 1; if the limit stored by (DO) is exceeded or reached, the loop end processing follows. Otherwise, jump back behind (DO) (see LOOP).

**(NUMBER)** ( d1 a1 - d2 a2 )
converts the ASCII sequence of digits from a1 + 1 accumulatively to d1. The result is d2.

a2 is the address of the first character, which is no longer convertible as a digit in the number system determined by BASE.

**))** ( -- )
switches back to the compile state within a word definition after it has been switched off with ((, consider. ((.

In FIG-FORTH 1

| | |
|---|---|
| ***** | ( n1 n2 -- n3 )<br>n3 = n2 * n1 |
| ***/** | ( n1 n2 n3 -- n4 )<br>n4 = n1 * n2/n3<br>The calculation takes place in the 32-bit number range, so it can still be used if the word sequence n1 n2 * n3 / would fail. |
| ***/MOD** | ( n1 n2 n3 -- n4 n5 )<br>n4 is the remainder, n5 is the quotient of the division (n1 * n2)/n3.<br>The calculation takes place in the 32-bit number range, so it is still applicable if the corresponding 16-bit calculation would fail. |
| **+** | ( n1 n2 - n3 )<br>n3 = n1 + n2 |
| **+ !** | ( na- )<br>n is added to the 16-bit number stored at addresses a, a+ 1. |
| **+ -** | (n1 n2 ~ n3 )<br>n3 = n1, if n2 > =0; n3 = -n1, if n2 <0. |
| **+LOOP** | ( n  - )<br>marks the end of a counting loop opened with DO. To be applied in the form:<br>n1 n2 DO Wordsequence n3 + LOOP<br>n1 = barrier, n2 = initial value, n3 = step size.<br>The loop is run through every time until the loop counter reaches or exceeds the n1 barrier, but at least once. The numbers n1, n2, n3 do not have to be specified explicitly, but can also be in the stack as calculated results.<br>Use:<br>Instead of DO or + LOOP, the compiler stores the code address of the associated runtime function (DO) or (+ LOOP). |

If the compiled function is run later, (DO) and (+ LOOP) work as follows:

(DO) writes n1 as a barrier and n2 as the initial value of the loop counter to the R stack; then the word sequence after (DO) is passed through. (+ LOOP) adds the increment n3 to the instantaneous value of the loop counter in the R stack.

If the limit n1 is exceeded, (or in the case of negative step width under steps) or reached, then loop end treatment takes place: n1 and n2 are deleted from the R stack, and the program is continued after (+ LOOP). Otherwise, a jump back to the beginning of the sequence of words stored after (DO) takes place.

**+ ORIGIN** ( n -a )
a = n + ORIGIN
(ORIGIN) is the address from which the FORTH system is stored. There are the initial values for the user variables and other system constants behind the jump commands on cold, warm and hot start. + ORIGIN is used to address them (see Appendix B).

( n - )
saves n from the next free dictionary address and increments the dictionary pointer DP by 2.

( n1 n2 - n3 )
n3 = n1 - n2

**-DUP** ( n - n n ) or ( n - n )
duplicates the top stack number if it is not zero. With -DUP, for example, the relatively frequent query:

Value DUP IF Word order OTHERWISE DROP ENDIF

can be simplified as follows:

Value -DUP IF Word order ENDIF

**-FIND** ( - a n1 n2 ) or ( - n3 )
brings the next word (completed with space) from the input string to the place HERE and searches for it in the CONTEXT and CURRENT dictionary.

If found, its parameter field address a, its length byte (including immediate bit) n1 and the identifier n2 = 1 (found) are stacked; otherwise, only the identifier n3 = 0 (not found).

**-TRAILING** ( a n1 -- a n2 )
a = Start address of a string (to be output with TYPE),
n1 = your length,
n2 = Length of the string after deleting the blanks on the right.

( n - )
Output of n in the number system set with BASE. Then a SPACE is output.

(FIG-FORTH)
Same as ." in PC-FORTH

**.CPU** ( -- )
displays the name LH5801 of the microprocessor used in the PC-1500.

**.LINE** ( n - )
brings the BASIC text line with the number n to the display. (If not present, error message ~ line n comes up).

In FIG-FORTH another meaning.

**.R** ( n1 n2 -- )
outputs n1 in a field of length n2 right-aligned.

**.S** ( - )
outputs all entries of the C stack as numbers without changing the stack contents. (Consider BASE.)

**."** ( -- )
Output of a text. To be applied in the form:

." Text"

Output the specified text. The end of the text is marked with" or by the end of the line.

The compiler stores the code address of the associated runtime function (.") and behind it the specified text with a length byte preceding it.

In FIG-FORTH ."

**/** ( n1 n2 - n3 )
n3 = n1/n2

**/MOD** ( n1 n2 -- n3 n4 )
n3 is the remainder, n4 is the quotient of the division n1/n2.

**0 1 2 3**

These small numbers were included as a constant in the dictionary, as they are used very often. In this way, only 2 bytes are ever needed in the compiled program and not 4 to represent these constants.

**0<** ( n1 - n2 )
n2 = 1 if n1 < 0, otherwise n2 = 0.

**0=** ( n1 - n2 )
n2 = 1 if n1 = 0, otherwise n2 = 0.
Among other things, applicable to the negation of a condition, therefore also sometimes referred to as DISTRESS.

**OBRANCH** ( n - )
Run-time function for IF, UNTIL and WHILE. Behind the code address of OBRANCH, the compiler stores the relative jump address. On expiration, a jump is performed when n = 0. Otherwise, the program will continue after the relative jump address.

**1+** ( n1 -- n2 )
n2 = n1 + 1

**2!** ( da-- )
stores the 32-bit value d from address a (4 bytes).

**2+** ( n1 - n2 )
n2 = n1 + 2

**2@** ( a -- d )
loads a 32-bit value d from address a onto the stack.

**2DUP** ( d -- d d )
duplicates the top d 32-bit value on the stack. Can of course also be used for
(n1 n2 — n1 n2 n1 n2).

**:**

initiates a word definition. Application in the form of:

: Name Wordsequence ;

The compiled word sequence is stored in the dictionary under the specified name. If the name is called later, the compiled word sequence is executed.

The compiler creates an entry in the dictionary of the following type:

| | | | | | |
|---|---|---|---|---|---|
| 1 | Name | Link | de | Compiled Wordsequence | se |
| T | | T | T | T | |
| NFA | | LFA | CFA | PFA | |

Length byte 1:

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Importance | 1 | im | sm | Name Length | | | | |

Name as indicated, but its last character is characterized by setting the most significant bit= 1.

link = start address of the last previously defined dictionary entry
de = Start address of the runtime function associated with the colon
se = code address of the runtime function belonging to semicolon; S
NFA = Name field address
LFA = Link field address
CFA = Code field address
PFA = Parameter field address

The colon builds this entry up to de

In the length byte, it sets the bit 7 to 1 to identify the length byte as such, and the bit 5 to characterize the definition as "still open". Bit 6 can be set by the word IMMEDIATE after completion of the definition; bit 5 is reset by the semicolon using SMUDGE. Finally, the colon sets the user variable STATE to Hex CO, so that the following words of the "word sequence" are not executed but compiled. (Exceptions, however, are the words marked as "immediate": these are also executed in the compile state.)

The runtime function associated with the colon causes the current state of the FORTH command pointer to be stacked in the R stack and the command pointer to be set to the beginning of the compiled word sequence. ;S performs the return to the calling function by deleting the top address in the R stack and saving it back to the FORTH command pointer.

**;**

ends a word definition (see description for :).

**;CASE**

used together with CASE: needed. Description see CASE:.
Not in FIG-FORTH.

**;S**

Run-time function for the semicolon (see description for :).

**<** ( n1 n2 -- n3 )
n3 = 1 if n1 < n2; otherwise n3 = 0

**<#**

starts processing a 32-bit number for the output.
This function is used together with # #S SIGN #>.

**<BUILDS**

Is used together with DOES> to generate new data types (such as String, array, etc.)

Application form:
: Typename <BUILDS Wordsequence1> Wordsequence2 ;

Typename is a name denoting the new data type;

Word sequence 1 is traversed when an object of the new data type is defined;
Word sequence 2 is passed through when the name of such an object is called up.

**< R >** ( n a- )
n is stored at address a, a + 1; previously, information is stored in the R stack, which causes the old content of a, a + 1 to be restored as soon as a return from the running function takes place.

Not in FIG-FORTH.

**< P** ( n - )
the effect is that the FORTH input is not made from the keyboard, but from the BASIC program memory from line N. PC FORTH reads line by line, ignoring the quotation mark at the beginning of each line. As soon as P> appears, the input is switched back to the keyboard.

If an error is detected when translating from the BASIC lines, a # character and the number of the incorrect line are added to the FORTH error message; in addition, the keyboard input mode is switched on like the. The incorrect definition may remain open and can be completed manually or deleted with (( SMUDGE FORGET Name.

If a non-existent program line is specified with n, the error message "Line ~ def #n" is displayed.

If a negative number is appended to an error message, the user has forgotten the final P>.

| | |
|---|---|
| **=** | ( n1 n2 - n3 )<br>n3 = 1, if n1 = n2; otherwise n3 = 0 |
| **>** | ( n1 n2 -- n3 )<br>n3 = 1, if n1 > n2; otherwise n3 = 0 |
| **> R** | ( n - )<br>n is cleared in the C stack and placed on the R stack. |
| **?** | ( a - )<br>The number n stored at the addresses a, a + 1 is output by '.' Abbreviation of @ . |
| **?COMP** | returns the error message "only in :Def" if the compile state is not enabled. |
| **?CSP** | outputs the error message "Structure error" if the stack level has changed compared to the stack level ensured in CSP. (Used by the compiler to ensure that word groups such as IF ELSE ENDIF can only be compiled in a structurally correct way.) |
| **7ERROR** | ( n1 n2 - )<br>The error message number n2 (BASIC line OFFSET + n2) is given if n1 is not = 0. |
| **7EXEC** | outputs error message "~ Compilable" if the execution state is not switched on. |
| **?LOADING** | returns error message "~loading" if the input from BASIC lines is not it is "loaded". Not in FIG-FORTH. |
| **?OI** | ( - n)<br>n = 1 if the user has terminated the last FORTH output stop with .; otherwise n = 0. Not in FIG-FORTH. |

**?PAIRS** ( n1 n2 - )

returns error message "does not fit" if n1 is i^n1. (Used by the compiler to ensure that different word groups such as IF ELSE ENDIF are not mixed in an inappropriate way.)

**7STACK**

outputs an error message "empty" or "full" when the stack boundaries are exceeded.

**?TERMINAL** ( - n )

n = 1 if the BREAK key was pressed during the execution of "pure" FORTH code; otherwise 0.

For FORTH input output via BASIC, the BREAK button leads to program interruption.

**@** ( a ~ n )

loads the 16-bit number n stored at addresses a, a + 1 onto the stack. Will be in the form:

Variblename @

used to load the contents of a variable onto the stack.

**@:** ( — a1 ) or ( — a1 n )

used to import numbers and strings from BASIC into FORTH. Used in word definitions that are read in by means of <P.

Application form:
@: "b"

where b is any BASIC expression (for example, the name of a BASIC variable) that does not contain a quotation mark.

If the specified BASIC expression b has a numeric result al, it is placed on the stack (— a1).

If the result of the given BASIC expression b is a string, then its initial address a2 and its length n are put on the stack (-- a2 n).

A numeric value is accepted only if it is in the range 0 to 65535, where digits after the decimal point are truncated. However, the calculation of b is carried out in the full range of BASIC numbers.

If b has a non-numeric result, the string is stored in the range that BASIC uses for this purpose. It will be overwritten as soon as BASIC processes another string (e.g. when outputting "OK"). If it is needed for a longer time, it must be stored (e.g. by CMOVE in the FORTH memory or by assigning it to a BASIC variable by means of !:). Not in FIG-FORTH.

If BASIC detects an error when processing b, FORTH gives the error message ?"basic.

For @:, the compiler stores the code address of the associated runtime function (@:) and the (basic encrypted) expression b with the preceding length byte and the following end separator.

**ABORT** (Deletes Stack)
executes warm start: Both stacks are deleted, the execution state is switched on, BASE is given the value 10, the message PC FORTH and the version designation are output.

**ABS** ( n1 ~ n2 )
n2 = absolute value of n1

**AGAIN**

to be used in the form:

BEGIN Wordsequence AGAIN

causes unconditional repetition of the sequence of words.

This loop cannot be exited, even with BREAK — unless it calls some function that doesn't necessarily return into the loop. (Example: CR, even if it is implicit. You can then give BREAK and cold or warm start.)

The compiler translates AGAIN into an unconditional jump to the place behind BEGIN.

**AIN** ( - a )

returns the address of the next character from the input string. See BLK.

**ALLOT** (n- )

adds n to the dictionary pointer DP (HERE). This allows storage space to be reserved in the dictionary or (since n may also be <0) stored later.

**AND** ( n1 n2 -- n3 )

n1 and n2 are linked bit by bit with a logical "And". A bit of n3 is set to 1 if and only if the corresponding bit in n1 and the corresponding bit in n2 are set (= 1).

**BAL** ( n- )

branches into BASIC to the program line n (n=1 to 255 ). The BASIC program, which is located there, is executed. The return to the FORTH takes place with

GOTO "H"

where the FORTH program continues at the point of interruption.

The BASIC variables OS, I$(0) and O$(0) used by FORTH must not be changed.

If the specified line does not exist, the error message ERROR 11 occurs in the BASIC.

Not in FIG-FORTH

**BACK** ( a - )

saves HERE-a as a 16-bit number from HERE. Used by the compiler to store real backward jump addresses. (The absolute jump destination address a is placed on the stack beforehand when compiling the word BEGIN or DO.)

**B#** ( n -- a )

Returns the start address a of the BASIC line with the number n. The address of the first character of this line is passed after line number and length byte. If no row with the number n exists, a = 0.

Not in FIG-FORTH

**BASE** ( -a )

The user variable BASE contains the valid number base for input and output conversions as a value.

After cold and warm start, BASE contains the value 10, so that the input and output of numbers is decimal.

**BEGIN** ( - )

See description UNTIL, WHILE, AGAIN

Compiler stacks the address HERE for the calculation of the relative return address, as well as a label for the structure check.

**BL** ( - n )

The constant BL stacks n = 32, that is, the ASCII value of the space (BLANK, SPACE).

**BLANKS** ( an- )

saves n spaces from address A.

**BLK** ( -a )

The user variable BLK contains the current line number as a value, while the FORTH input is made from lines of the program memory.

n <P stores the line number n at the address a of the user variable BLK. If a line has been processed, the number of the next line comes after a. If a line ends with P>, the 16-bit number 0 is stored at a. This 0 is also the indicator for the manual input mode.

In FIG-FORTH another meaning.

**BR** ( -a )

The user variable BR contains the CPU stack level (i.e. the value of the S register) as a value before calling the FORTH system. This value is returned to the S register each time it is returned from the FORTH system.

Not in FIG-FORTH

**BRANCH**

causes unconditional jump. When processing ELSE, AGAIN and REPEAT, the compiler stores the code address of BRANCH and the relative jump destination address behind it.

**C!** (na- )

stores n (n<256) at address A. (Only the low-value 8 bits of n are stored at address a, the content of a + 1 is not changed.)

**C#** ( n1 - n2 )

converts a numeric value n1 to the corresponding ASCII code n2. BASE is not considered. So:

n2 = n1 + 48, if 0 ^ n1 S 9
n2 = n1 + 55, if 10 S n1 2 200

Not in FIG-FORTH

**C,** ( n - )

The low-order 8 bits of n are stored at address HERE. C, so stores 1 byte in the dictionary, HERE is increased by 1.

**C@** ( a -- n )

loads the contents of the memory cell with the address a onto the stack, n is the value of the byte stored there (n < 256).

**CASE:**

is used to define "distribution functions" and is to be applied in the following way:

CASE: ex c0 c1 c2 .. cn ;CASE

In this case, c0 to cn must be the names of previously defined FORTH words; i.e., the desired name of the distribution function to be defined must be added. (This may be new.)

After this distribution function ex is defined in this way, it can be called manually or in word definitions.

If the function ex is to be executed, then at the top of the stack there must be a number n1, which has one of the values 0 to n and indicates which of the functions c0 to cn should be called.

The action of the function ex consists in popping the number n1 off the stack and then calling the n1th function cn1 (as the only one). After cn1 has returned, processing continues after ;CASE.

Not in FIG-FORTH

**CFA** ( a1 -- a2 )
CFA converts the parameter field address a1 into the associated code field address a2. In PC-FORTH, a2 = a1-2.

**CLS** (- )
clears the output buffer and sets the output pointer OUT to 0.

Not in FIG-FORTH

**CMOVE** ( a1 a2 n - )
copies n bytes from address a1 to a2.

In the case of overlapping areas, it is important that the content of a1 is copied to a2 first, then the content of a1 +1 to a2+1 etc.

(If this mode of operation is undesirable, CX is used)

**COLD** (initialized stack)
causes cold start. Deletes all user definitions, initializes values in the user variables and then executes warm start (ABORT).

**COMPILE**

In contrast to ((COMPILE)), it is translated and stored by the compiler as normal and has no effect on the current activity of the compiler.

However, if the function in whose definition COMPILE appears is called later, the translated word after COMPILE is not executed, but is stored in the dictionary from HERE onwards.

**CONSTANT** ( n -- )

creates a constant with the value n. Apply it in the form:

n CONSTANT name

When the name is called later, the value n is placed on the stack.

**CONTEXT** ( - a )

The value of this user variable refers to the dictionary in which the entered words should be searched first.

**COUNT** ( a1 -- a2 n )

COUNT expects the start address a1 of a text entry beginning with a length byte as the top stack entry; a2 = a1 + 1 is the starting address of the actual text; the low-order byte of n is the value of the length byte, the high-order byte of n = 0. (The two values a2 and n are thus expected by TYPE on the stack.)

**CR** (- )

causes the actual output of the characters that were placed (by EMIT, TYPE, ., .", etc.) in the output buffer.

The output takes place in BASIC in line 38. The printer is controlled with the basic variable L.

If the printer is switched on (L = 1), the output takes place on the printer and display. Then, if no key is pressed, the program execution is continued, otherwise the output is stopped.

If the printer is switched off (L = 0), the output is only on the display. It then stops output and waits for a key to be pressed, after which program execution continues. If the I key is pressed, the TRACE is switched on; the TRACE can be switched off again with the T key.

**CREATE**

is applied in the form:

CREATE Name

to create a name entry in the dictionary. The name entry has the following form:

1 Name link cfa

The length byte 1 contains the length of the name in the lower 5 bits (bit 0 ... bit 4), bit 5 to bit 7 are set to 1. The name is saved as specified; only its last character is identified by a 1 in bit 7, link is the start address of the most recent previously defined name entry, cfa is the address of the following byte.

**CSP** ( - a )
The user variable CSP is used by the compiler for structure checks. Its value is the current stack position after opening a word definition. It must match the stack level at the end of the definition (using a semicolon); otherwise "structural error".

**CSWAP** ( n1 ~ n2 )
swaps the high-order byte of n1 with the low-order byte, i.e. the top two bytes in the stack.

Not in FIG-FORTH.

**CURRENT** ( - a )
The content of the user variable CURRENT points to the vocabulary in which the current definitions are entered.

Initially, this is the FORTH vocabulary. However, DEFINITIONS can be used to activate the CONTEXT vocabulary for entering definitions.

**CX** ( a1 a2 n - )
swaps the memory area from address a1 with the memory area from a2 with a length of n bytes. In the case of overlapping areas, it is important that the content of the last byte in the first area is first swapped with the content of the last byte in the second area, then the content of the penultimate bytes, etc. Processing is therefore in the opposite direction to CMOVE

Not in FIG-FORTH.

| | |
|---|---|
| **D+** | (d1 d2-d3 )<br>d3 = d1 + d2 |
| **D+-** | (d1n--d2 )<br>d2 = d1, if n > 0 or n = 0; otherwise d2 = -d 1 |
| **D.** | ( d - )<br>returns a double-precision (32-bit) number. The higher-order 2 bytes are expected at the top of the stack. To get the double number Hex 11223344, you can enter HEX 3344 1122 D. |
| **D.R** | ( d n - )<br>outputs a double-precision number d right-aligned in a field of length n. |
| **DABS** | ( d1 -- d2 )<br>d2 = Absolute value of d1 |
| **DEC** | ( - )<br>the value of BASE is set to 10, so that the input and output of numbers are decimal. Called DECIMAL in FIG-FORTH. |
| **DECIMAL** | (FIG-FORTH)<br>In PC-FORTH see DEC |
| **DEFINITIONS** | sets the value of CURRENT to the value of CONTEXT.<br>This adds new definitions to the CONTEXT vocabulary in which the compiler (using -FIND) searches (first) for the names to be translated. |
| **DIGIT** | (n1 n2 - n3 n4, if feasible)<br>( n1 n2 - n4, if not feasible)<br>Conversion of a digit in the ASCII code into the corresponding numerical value. |

n1 = ASCII value of the character
n2 = Base of the number system to be used
n3 = resulting value
n4 = 1 if conversion is feasible; n4 = 0 if conversion is not feasible

**DLITERAL** ( d - )
During compilation, translates a previously stacked double-length number so that when the generated FORTH program code is run, this double-length number is pushed onto the stack. The generated program piece is basically:

LIT lower half LIT upper half

where, of course, instead of LIT, the code address of LIT is stored and both halves of the double number are binary-encoded (see LITERAL).

**DMINUS** ( d1 - d2 )
d2 = -d1 (in two's compliment)

**DO** ( n1 n2 -- )
opens a counting loop with the initial value n2 and the limit n1. Used in the two phrases

DO .. LOOP
DO .. +LOOP

For description see LOOP or +LOOP.

**DOES>** ( a -)
Used in the words:

<BUILDS .. DOES> ..

For description see <BUILDS.

| | |
|---|---|
| **DP** | ( -a )<br>The value of this user variable is the Dictionary pointer, it contains the address of the first free byte in the dictionary. The value of DP can be put on the stack with HERE and changed with ALLOT. |
| **DPL** | ( -a )<br>The value of this user variable is set when converting a number present as an ASCII sequence of digits, as is done, for example, when entering numbers. It contains the number of digits after the decimal point. If no decimal point is found, the number is converted as a simple exact value and DPL = -1 is set. If a decimal point is found, a double-exact value is created, and DPL is given a value > 0.<br>FORTH works internally only with integers. With the help of DPL, however, the user can handle fixed-point numbers himself, as is useful, for example, for amounts of money in DM. |
| **DROP** | ( n - )<br>The top number in the stack is deleted. |
| **DUMP** | ( an- )<br>returns a memory range of n bytes length starting at address a by means of . from. |
| **DUP** | ( n - n n )<br>duplicates the top number in the stack. |
| **ELSE** | ( - )<br>is used in the word order<br>.. IF .. ELSE .. ENDIF is needed. See IF. |
| **EMIT** | ( n - )<br>writes a character to the output buffer. The low-order byte of n is interpreted as the ASCII value of the character to be output. The output pointer OUT is incremented by 1. If the value of OUT reaches 24, EMIT appends the character ~ as an output separator and emits the output buffer. |

**ENCLOSE** ( a n1 - a n2 n3 n4 )

ENCLOSE means "to include". This function is used, for example, by the word reading function WORD to determine the next FORTH word in the input string and to "enclose" it in pointers (i.e. set pointers to its beginning and end).

With ENCLOSE, however, numbers, comments, texts and BASIC expressions inserted in FORTH are also "enclosed".

a = Address from which the ENCLOSE function should search
n1= delimiter (ASCII encoded), e.g. 32= Blank, 41= ")", etc.
n2 = distance from a to the first character of the found word (generally after reading over preceding blanks or separators)
n3 = distance from a to the 1st character after the found word
n4 = distance from a to the address from which the ENCLOSE function is to search at the next call.

As in the original FIG-FORTH, ENCLOSE also considers a binary zero as an (exceptional) separator; in contrast to FIG-FORTH there is another exceptional separator, namely the BASIC end-of-line character (ENTER), whose ASCII code is 13. This is to ensure that when entering FORTH source code from BASIC lines, the line number is not also translated, but is kept in BLK (for any necessary error messages).

**END** ( n - )

equivalent to UNTIL

**ENDIF** ( -- )

used in the groups of words .. IF .. ELSE .. And ENDIF .. IF .. Used ENDIF and described at IF

**ERASE** ( an-- )

deletes a memory area of n bytes length from address a by writing over with zero

**ERROR** ( n - )

used to output error messages.

However, if WARNING has the value -1, the message is suppressed and a warm start (ABORT) with deletion of both stacks is initiated. The user can (with appropriate caution) provide his own error handling function to call instead of the warm start.

If the value of WARNING = 0, the name of the function in which the error was detected and the error number n (preceded by a question mark and quotation marks) are output. If WARNING has the value 1, the BASIC line with the number offset + n is output as the error text instead of the error number (where offset is the value of the user variable OFFSET).

If this line to be output is not defined, the message "~ line" and the number of the undefined line will be displayed instead of the text; the user can calculate the error number by subtracting the value of OFFSET.

**EXECUTE** ( a - )
executes the function whose code field address a is at the top of the stack.

**EXPECT** (an- )
fetches up to n arbitrary characters from the keyboard, saves them from address a and outputs them at the same time. EXPECT is terminated either when the n+1th character is entered or prematurely when a control character is entered. (A terminating control character is an ASCII code<32 character, excluding the shift characters DEF, SHIFT, SML.)

In any case, a binary zero is appended to the stored characters (so that the area to be kept free must be greater than n by 1); and the terminating character is stored as the value of the user variable OIV, where it is available for flow control that may be desired.

(In FIG-FORTH, ENTER is the only character that can end the EXPECT input prematurely; the effect of the other control characters is not explicitly specified. It is also unclear how many binary zeros are appended as end delimiters.)

**FENCE** (~a )
The value of the user variable FENCE ("Fence") acts as the lower Bound for the FORGET function

Words whose name entries begin below address a cannot be deleted with FORGET (or only after the value of FENCE has changed). However, cold start ignores FENCE and erases all user-defined words.

**FILL** ( a n1 n2 --)
n1 bytes from address a are assigned the value n2 (low-order byte of n2).

**FORGET**

deletes definitions from the dictionary. Application in the form of:

FORGET Name

All definitions made after "name" are "forgotten". The most recent definition of the specified name and all more recent definitions are deleted and the space in the dictionary is freed up again.

However, if the entry of the specified name begins at an address less than the value of FENCE, an error occurs; nothing is deleted, and the starting address of the name entry is pushed onto the stack.

**FORTH**

is the name of the vocabulary that is available from the start. As long as the user has not defined any other vocabulary, all of his definitions are included in the FORTH vocabulary.

If the user has declared another vocabulary as the CONTEXT vocabulary and (after DEFINITIONS) has entered functions in it, he can (by calling FORTH again declare the FORTH vocabulary as the CONTEXT vocabulary

With DEFINITIONS, the CURRENT vocabulary is then also equated with the CONTEXT vocabulary.

The user then works exclusively in the FORTH vocabulary; the names in the other vocabularies are not deleted, but are neither displayed nor found in the dictionary search. They are only available again when the user calls up the name of the other vocabulary.

**HERE** ( -a )
pushes the value a of the dictionary pointer DP onto the stack, a is the address of the first free byte in the dictionary.

**HEX** ( - )
activates the hexadecimal number system for the input and output of numbers by giving BASE the value 16. With DEC you can switch back to decimal number representation, as is done automatically with a warm or cold start of FORTH.

**HLD** ( - a )
While a number is prepared for output with < # # #S, the user variable HLD contains the address at which the last converted character was stored. Initialized by < #.

**HOLD** ( n -)
can be used during output editing of numbers between < # and #> to append a decimal point or other character to the already edited digits. Then the remaining digits can be processed, n is the ASCII code of the character to be inserted.

**I** ( - n )
The function I can be used in DO loops to copy the current count n of the loop counter from the R stack to the C stack.

**ID.** ( a - )
outputs the name of the FORTH word whose name entry starts at address A.

**IF** ( n - )
Application only within word definitions in the form:

Condition IF Wordsequence1 ELSE Wordsequence2 ENDIF or

Condition IF phrase 1 ENDIF

If the condition is met (general: if IF finds a value other than 0 at the top of the stack at execution time), wordsequence1 is run through, otherwise wordsequence2, if present. In both cases, processing continues after ENDIF. Nesting is allowed.

The compiler translates IF into a conditional jump OBRANCH to the place after ELSE or (if ELSE is missing) after ENDIF. For ELSE, an unconditional branch BRANCH is compiled to the position after ENDIF.

**IMMEDIATE** (--)

Sets the "immediate bit" in the most recent name entry in the dictionary. This means that if the name marked in this way is called later, its function will be executed immediately, even if this happens at compile time. IMMEDIATE can therefore be used to extend the compiler with additional functions.

**IN** (-- a )

The value of the user variable IN is the number of the character in the input buffer that is to be processed next. The number runs from 0 to a maximum of 79.

**INS** ( - a)

The address INS, defined as a constant, indicates the position in the FORTH system where the currently connected trace function TRC is called. INS is used by the INSERT function.

Not in FIG-FORTH.

**INSERT**

This function allows connecting a user-defined trace function (with any name xxx) instead of the built-in trace function TRC. After the user has defined his trace function xxx:

INSERT xxx

he can connect it to the FORTH system. It is then called before each FORTH word when FORTH functions are run through with the trace mode switched on. The address at which the relevant FORTH word is called is transferred at the top of the stack and must be deleted in function xxx.

If the user later wants to connect the original TRC function again, he can do so with:

INSERT TRC

On the other hand, if he no longer needs the original TRC function, he can delete it (after changing FENCE). If the user does not need any trace function at all, it is recommended to use

INSERT .

to connect the number output function instead of TRC. This ensures that the FORTH system does not crash if the l key is accidentally pressed. You can switch off this "pseudo-trace" with T and continue the program with ENTER.

Not in FIG-FORTH

**INTERPRET**

This function is called by QUIT after QUIT uses QUERY to fetch a new input string from the keyboard.

INTERPRET fetches word by word from the input using -FIND until it is processed. If a word is found in the dictionary, it is either compiled or (compiled and) executed (depending on the immediate bit and the position of the switch in STATE). If it is not found, an attempt is made to process it as a number (according to the set base). If that is not possible either, an error message ("— def") follows and the interpretation process is aborted. In some way (OK message, error message, warm start, ...) the program continues back to QUIT, where the next input character string is fetched and INTERPRET is called again.

**KERN**

Deletes (without regard to FENCE) all non-essential FORTH functions except for the "core". The deleted functions are made available on the cassette as FORTH source code in BASIC text lines and can (after interrupting the running FORTH core system) be used with the BASIC command

CLOAD "PC-FORTH. SRC"

(see Appendix B).

After cold start using DEF C, the functions can be added with:

40 <P

to the FORTH core again. This creates exactly the original PC-FORTH.

Instead, the user can (after reading in the source code from the cassette) switch on the BASIC programming mode and make changes in the FORTH source code before adding the functions again (using <P in FORTH).

We have removed as many functions as possible and sensible from the "core" in order to give every user a free hand in designing their own personal FORTH system. However, we must point out that if you want to exchange programs with other FORTH programmers, you must exercise appropriate caution and restraint when making changes.

Not in FIG-FORTH.

**KEY** ( - n )
fetches a character from the keyboard and pushes its ASCII value n onto the stack.

**LATEST** ( - a )
The value of this user variable is the starting address of the most recent name entry in the CURRENT vocabulary.

**LEAVE** ( - )
is applied between DO and LOOP or between DO and +LOOP and sets the bound (put on the R stack by DO) to the current value (also put on the R stack) of the loop counter. This allows the loop to be exited the next time LOOP or +LOOP is called.

**LFA** ( a1 - a2 )
converts a parameter field address to a link field address.
In the PC FORTH, a2 = a1 – 4

**LIT** ( -- n )
If the compiler finds a number n in the source code, it first stores (the code address of) LIT and then the (binary-coded) number. If LIT is called later, the 2-byte number n stored after LIT is placed on the stack.

**LITERAL** ( n - )
During compilation, translates a number that was previously put on the stack so that this number is later put on the stack when the generated program snippet is run through

Typical application:
:...(( Calculate number n )) LITERAL... ;

In principle, the translated program piece looks like this:
... LITn ...

(where, of course, instead of LIT, the code address of LIT is stored and n is binary encoded in the two subsequent bytes).

For example, calculations whose results are already known at compile time can also be carried out during compilation. The translated function immediately contains (after LIT) the finished, binary-encoded result, just as it should be placed on the stack when the translated function is called.

**LOOP** ( - )
Application within word definitions in the form:

n1 n2 DO word sequence LOOP
n1 = bound; n2 = initial value of the loop counter

The sequence of words between DO and LOOP is repeated until the Loop counter has reached or exceeded the limit n1, but at least once.

The compiler stores the associated runtime function (DO) at the location of the DO call, and the runtime function (LOOP) at the location of LOOP, followed by the "relative backward jump address".

At runtime, (DO) pushes the top two numbers n2 and n1 from the C stack to the R stack. Then the word sequence is run through for the first time.

LOOP adds a 1 to the loop counter n2 stored in the R-stack and then compares it with the bound n1. If the bound has not yet been reached, the return jump takes place according to the relative backward jump address stored after LOOP.

Otherwise, LOOP clears n1 and n2 from the R-stack and processing continues past where the backward relative jump address is stored.

See also +LOOP.

**M*** ( n1 n2 - d )
d = n1 * n2

**M/** ( d n1 - n2 n3 )
n2 = remainder of division d/n1
n3 = quotient of division d/n1
The sign of n2 always agrees with the sign of d.

**M/MOD** ( ud1 a1 -- a2 ud2 )
performs mixed division:
ud1 = dividend (unsigned, double length)
a1 = divisor (unsigned, single length)
a2 = remainder (unsigned, single length)
ud2 = quotient (unsigned, double length)

**MAX** ( n1 n2 - n3 )
returns the maximum of two numbers: n3 = Max(n1,n2)

**MESSAGE** ( n -- )
prints the BASIC line with the number n + the value of OFFSET as a message if WARNING does not have the value 0. If the BASIC line is not defined, the message "~ line" followed by the line number is output.

If the value of WARING = 0, then n itself is output (identified as a message by preceding quotation marks).

**MIN** ( n1 n2 - n3 )
returns the minimum of two numbers: n3 = Min(n1,n2)

**MINUS** ( n1 - n2 )
n2 = -n1 (two's complement)

**MOD** ( n1 n2 -- n3 )
n3 = remainder after dividing n1/n2
(n3 always has the same sign as n1)

**NFA** ( a1 - a2 )
calculates the associated name field address from a parameter field address. (Uses TRAVERSE)

**NOOP** ( - )
does nothing (No Operation).

**NUMBER** ( a - d )
converts the character string stored from a with a preceding length byte into a double-precision number d, respecting BASE. As stated in the original FIG-FORTH, the high byte of the number is at the top of the stack.

Negative double numbers are represented in two's complement. If a decimal point is found during conversion, the function stores the number of digits after the point in DPL; otherwise DPS gets the value -1 .

If the conversion is not possible, an error message appears.

**OFFSET** ( -a )
The value of this user variable specifies the number of the BASIC line from which the error message texts are saved. If WARNING has a value < 1, the value of OFFSET is irrelevant.

Different meaning in FIG-FORTH.

**OIV** ( -a )
(Abbreviation for "Output Interrupt Variable")

The value of this user variable contains the character with which the user ended the last output stop or EXPECT call.

**OR** (n1 n2 - n3 )
n3 = n1 OR n2
Each bit in n3 is set if and only if the bit of the same value is set in n1 or in n2 (or in both numbers).

| | |
|---|---|
| **OUT** | ( - a )<br>This user variable contains a value that is increased by 1 for each character that is output (with EMIT).<br>In PC-FORTH, OUT counts modulo 24, i.e. after 24, the output buffer of EMIT is output and OUT is set to zero. |
| **OVER** | ( n1 n2 – n1 n2 n1 )<br>pushes a copy of the second from top number on the stack, on top of stack. |
| **P >** | ends the input from the BASIC program memory (see <P).<br>Not in FIG-FORTH. |
| **PÄD** | ( - a )<br>This function returns the start address of a memory area that can be used as a temporary buffer. The address supplied by PÄD is always 68 bytes after HERE, so it changes when words are defined or deleted. |
| **PFA** | ( a1 - a2 )<br>converts the name field address of a name entry into the parameter field address. (Uses TRAVERSE) |
| **PICK** | ( n1 -- n2 )<br>copies the n1th number from the stack to the top of the stack.<br>n2 is the n1th number in the stack, counting from the top, but not counting n1.<br>PICK does not check whether the searched entry is still within the stack area. |
| **QUERY** | requests a string from the keyboard. For this purpose, PC-FORTH switches to BASIC and requests input with INPUT I$(0). Therefore, typing errors can be corrected during input, as usual with BASIC. The entry is completed with ENTER. The user variable TIB supplies the start address of the entered text. |

I$(0) is dimensioned as an 80-byte text field. Its start address is in the user variable TIB; it is set there on a cold start from ORIGIN + 20.

The input buffer TIB must not normally be changed in user programs, so QUERY should only be used with great caution in user programs.

**QUIT**

Clears the R stack and ends input from BASIC program lines if necessary. However, any existing compile state is not terminated. QUIT does not issue a message.

**R** ( - n )

copies the top number from the R-stack to the C-stack. (Thus, R acts exactly like I.)

**R>** ( -n )

moves the top number from the R-stack to the C-stack. (So, the top R-stack entry is deleted.)

**RO** ( - a )

This user variable contains the initial value of the R stack level (address where the R stack begins). See Appendix B

**REPEAT** ( -- )

Will be in the form:

BEGIN .. WHILE .. REPEAT

Use, see WHILE.

**ROT** ( n1 n2 n3 - n2 n3 n1 )

moves the third top number in the stack to the top, pushing the top and second top numbers down.

**RP** ( -a )

This user variable contains the level of the R stack.

**RP!**

writes the value of user variable RO to user variable RP and uses it to initialize the R stack

**RPICK** ( n1- n2 )

copies the n1th number of the R-stack (counting from the top from 1) onto the C-stack, overwriting n1 on the C-stack. (n2 will generally have the meaning of an address.)

**S-> D** ( n -- d )

converts a simple number to a double-length number equal to it.
(Depending on the sign of n, the upper half of the 4-byte double number is either Hex 0000 or Hex FFFF.)

**SO** ( - a )

This user variable contains the initial value of the C-stack level
(Address where the stack starts). See Appendix B.

**SIGN** ( n d - d )

is applicable during the output conversion of a number, i.e. between < # and #>. Puts a minus sign in front of the previously converted string if n< 0. n is deleted, but the 4-byte number d remains unchanged because it is still needed during the conversion.

**SMUDGE**

In the most recent name entry, the "smudge bit" (in the length byte) changes from 0 to 1 or from 1 to 0. As long as a definition is not complete, its name should not normally be found by -FIND. This is achieved by setting the "smudge bit". However, recursive programs require a function to call itself. To do this, SMUDGE is used in the definition before the recursive call.

**SP!** (clears stack)

writes the value of the user variable to the register S of the microprocessor and thus sets the stack level to its initial value.

| | |
|---|---|
| **SP@** | ( - a )<br>Pushes the current stack level onto the stack as address a, (so a is the address that was at the stack level before a was stacked.) |
| **SPACE** | ( - )<br>outputs a space. |
| **SPACES** | ( n -- )<br>outputs n spaces. |
| **STATE** | ( -a )<br>The value of this user variable is 0 during run state and hex CO during compile. |
| **SWAP** | ( n1 n2 -- n2 n1 )<br>swaps the top two stack entries. |
| **TASK** | is often defined as the boundary between different groups of related user functions. With FORGET TASK, the user can then delete the most recent group(s) in each case. Is only for clarity, will not be called. |
| **THEN** | works exactly like ENDIF. (Do not confuse this with THEN from BASIC!) |
| **TIB** | ( - a )<br>(Abbreviation for "terminal input buffer")<br>The value of this user variable is the start address of the input buffer. It must be identical to the BASIC variable I$(0). |

**TIC** ( -- a )
To be used in the form:

TIC name

Returns the parameter field address (PFA) of the definition of the word "name". In general, the compiled code of the word definition is stored starting at a.

In FIG-FORTH '

**TOB** ( -a )
(Abbreviation for "terminal output buffer")
The value of this user variable is the start address of the output buffer. It must be identical to the BASIC variable O$(0).

Not in FIG-FORTH.

**TOGGLE** ( an- )
Only the low-order byte of n is used. It is "XORed" with the byte at address a. The result byte is stored in a.

**TRACE**

turns on the trace switch. This causes the TRC function to be called before each FORTH word to be called. Thus, TRACE acts like 1 (but 1 can only be given during an output step).

The trace switch can be switched off again with TROFF or T and the program run can be continued.

TRACE and TROFF should only be given in word definitions. They also work if entered manually, but then the compiler and interpreter themselves are monitored step by step.

EXECUTE cannot be properly monitored with TRC; therefore, TROFF or T must be given before EXECUTE.

Not in FIG-FORTH. Not in FIG-FORTH.

**TRAVERSE** ( a1 n - a2 )

used in NFA and PFA to find either the next preceding or next following byte with bit 7 set, n indicates direction of search. n is either +1 or -1 and is added to a1 until the content of the resulting address is again a byte with bit 7 = 1. The resulting address is left as a2.

**TRC** ( a - )

When Trace is switched on, a call to the TRC function is inserted before each FORTH word call. The address a, from which the relevant function call (i.e. the code address of the FORTH word to be called) is stored, is passed to the function TRC as the top stack entry.

TRC now outputs the following:
the address a
the name of the FORTH word to be called
the top stack entry
the second top stack entry and
a CR

The numbers are output in unsigned hex format.

ENTER or I continues Trace; T switches off the trace so that the program run can continue normally.

TRC was programmed without much effort; however, the existing weaknesses should normally be acceptable. (If not, see INSERT)

TRC has the following disadvantages:

— Monitoring of EXEC is not performed properly. As soon as EXEC is issued, the trace must be switched off.
— It is not possible to look at variables or other memory contents using additionally inserted FORTH functions. (For this purpose it would have been necessary, e.g. the content of the input buffer.)
— The content of the output buffer is overwritten by the trace output.
— TRC also monitors the functions of the FORTH system.
Therefore, before words like . or D. switch off the trace.

Not in FIG-FORTH

**TROFF**

turns off the trace. It thus has the same effect as t.

Not in FIG-FORTH.

**TYPE** (an- )

outputs the text of length n stored from a.

**U*** ( a1 a2 - ud )

ud = a1 * a2 (U * is the multiplication routine that all other multiplication instructions fall back to and is by far the fastest to run.)

**U.** ( a - )

returns the topmost stack entry unsigned, taking BASE into account.

**U/** (ud a1 -- a2 a3 )

divides ud by a1, a3 is the quotient, a2 is the remainder. (U/ is the division routine that all other division commands use, so it's by far the fastest to run.)

**U <** ( a1 a2 -- n )

1 if the relation al < a2 applies to the two unsigned numbers al and a2, otherwise n = 0.

**UNTIL** ( n -)

is used within definitions in the following form:

BEGIN phrase condition UNTIL

The sequence of words is repeated until the condition is met (i.e. until UNTIL finds a number other than 0 as the top stack entry).

The compiler translates UNTIL as (code address of) OBRANCH with subsequent relative backward jump address.

**USER** ( n- )

defines a user variable. call in the following form:

n USER Name

where n specifies the distance to the start address of the user area (in bytes).

User variables and user areas are only relevant in multitasking systems in which several users work with different user areas at the same time

Since access to normal variables is much faster than access to user variables, the user area in PC-FORTH was designed in such a way that a maximum of 7 new user variables can be defined. If you do not want this restriction, change the start address of the user area on ORIGIN + 14 before the cold start so that the desired amount of memory space is available (see Appendix B).

A maximum of 128 user variables can be defined in total (in the entire FORTH system). (Same as in FIG-FORTH.)

**VARIABLE** ( n - )
in the following form:

n VARIABLE name

used to define a variable with the specified name and assign it the top number n on the stack as its initial value, n is popped from the stack. Calling the name later causes the address from which the value of the variable is stored to be placed on the stack. The value can then be loaded onto the stack with @ or with! to be changed.

**VOC-LINK** ( - a )
The value of this user variable points to the name entry of the most recent (i.e., most recently defined) vocabulary. With its help, the FORTH system concatenates all vocabularies.

**VOCABULARY**
To be used in the form:

VOCABULARY Name IMMEDIATE

defines a vocabulary with the specified name. Calling up the name later makes the vocabulary the "context vocabulary", i.e. the vocabulary in which INTERPRET first searches for all the words entered.

In PC-FORTH (as in FIG-FORTH), the vocabularies are concatenated such that if a vocabulary is searched unsuccessfully, the search continues in the vocabulary in which the unsuccessful searched vocabulary was defined. As a result, the words of the FORTH vocabulary are always within reach.

**VLIST** ( - )
outputs all names defined in the context vocabulary. After processing a vocabulary, the vocabulary follows in each case by defining the processed vocabulary.

Output can be aborted after output has stopped with the • key.

**W,** ( n - )
works like WORD, but also stores the word delimited by the character with the ASCII code n as a character string from HERE in the dictionary, with DP (HERE) being incremented accordingly. Not in FIG-FORTH.

**WARNING** ( - a )
If the value of this user variable is 1, error messages are output as text.

If it is 0, only error numbers are output.
If it is -1 , ABORT always occurs in the event of an error.

The error message texts are in BASIC lines and can (in BASIC programming mode) be changed by the user as desired.

The BASIC line numbers of the error message texts are calculated from error number + value of OFFSET; if the texts are moved to a different line number range, the value of OFFSET must be changed accordingly.

All error texts can be deleted to save space. To do this,
set the value of WARNING to 0 (or -1).

**WHILE** ( n - )

To be used in the form:

BEGIN Wordsequence1 Condition DURING Wordsequnce2 REPEAT

Wordsequence1 is run through at least once. Otherwise, Wordsequence1 and Wordsequence 2 are repeated as long as the condition is met (i.e., as long as WHILE encounters a non-zero value at the top of the stack).

Then processing continues after REPEAT.

The compiler does not store any code for BEGIN, it only remembers the position. For WHILE it saves a conditional jump to the word sequence entered after REPEAT (ie OBRANCH and relative forward jump address), and for REPEAT an unconditional return to the word sequence (i.e. BRANCH and relative backward jump address).

**WIDTH** ( ~a )

The value of this user variable specifies the maximum length of names entered in the dictionary. By default, WIDTH contains the value 31. A lower value of WIDTH saves storage space, but Forth names are then only signed with a smaller number of characters. Some FORTH systems work with WIDTH = 3.

**WORD** ( n - )

reads a word delimited by the character with ASCII code n from the input string into memory starting at address HERE.

The length byte is stored first, then the character string followed by one or more spaces.

**WLIST** ( a - )

serves to list the FORTH code generated by the compiler. Typical application in the form:

TIC Name WLIST

outputs the addresses, the code addresses stored there and the associated FORTH names from address a (TIC name) onwards.

When output is stopped, • can be used to abort WLIST.

WLIST works in the same way as a disassembler and is therefore faced with similar problems when retranslating the code: Data stored in the code may be misinterpreted and, if the number of data bytes is odd, can lead to WLIST "unhooking" and incorrect output.

| | |
|---|---|
| **XOR** | ( n1 n2 - n3 )<br>n3 = n1 XOR n2. An Exclusive OR is performed bitwise between n1 and n2. |
| **[** | (FIG-FORTH)<br>is called (( in PC-FORTH |
| **[COMPILE]** | (FIG-FORTH)<br>is called ((COMPILE)) in PC-FORTH |
| **I** | (FIG-FORTH)<br>is called )) in PC-FORTH |

# Chapter 4

## Example: Useful definitions

Below are some useful word definitions to help you work with the Forth system. However, these definitions should primarily serve to suggest how you can further expand PC-FORTH.

## 4.1 Printer control

First, some definitions concerning the printer:

```
200": LON(- )
205" (turns printer on)
210" 1 !: "L" ; (L=l )
215"
220": LOFF(- )
225" (turns printer off)
230" 0!:"L" ; (L = 0)
235"
240"HEX
245": CSIZE (n-- )
250" 79F4 C! ;
255"DEC
260": COLOR (n- )
265" !:"C" (C = n)
270" 26 BAL ; ( GOTO 26)
275"P>
```

Line 26 in BASIC is entered for use by COLOR as follows:

```
26 COLOR C:GOTO"H"
```

With LON and LOFF you can programmatically switch the printer on and off. The actual printer control takes place via the BASIC variable L in lines 20 to 25 of the BASIC part of the Forth system. CSIZE and COLOR correspond to the BASIC commands.

## 4.2 Miscellaneous

The following two commands are used for memory management. MEM returns the currently occupied memory areas. DELETE is used to delete BASIC lines in the program memory.

```
300"HEX
305": MEM ( – )
310" (shows memory usage)
315" .'FORTH: – 0 +ORIGIN 6 .R HERE 6 .R CR
320" ." STACK: " SP@ 6 .R SO 6 .R CR
325" ." BASIC: " 7865 @ 6 .R 7867 @6.RCR ;
330"
335": DELETE (n– )
340" (delete all BASIC lines from number n)
345" B# 3 – DUP (1st byte of line number)
350" FFSWAPC! (End marker FF)
355" 7867 ! ; (points to end of program memory)
360"P>
```

The following words are for the cooperation between BASIC and PC-FORTH. With GOTO a BASIC line with any number can be started, with the BASIC command a BASIC program that connects directly to the Forth source code can be started.

```
400": GOTO ( n – )
405" (jumps to BASIC line n, return with GOTO "H")
415" !:"G"40BAL; ( n after G and jump to line 40)
```

Line 40 is entered like this:

```
40 GOTO G
```

```
420": BASIC (– )
425" (changes from Forth to BASIC)
430" DROP 0 IN ! (Remove relics of <P)
435" BLK @ ( current line number )
440" 0 BLK ! (cancel from <P )
445" GOTO ;
```

Line 41 is entered as follows:

```
41"FORTH":I$(0) = STR$(I) + " <P":GOTO "H"
```

Now BASIC and Forth statements can be mixed like this:

```
1000 PRINT'We're in BASIC!'
1010 1= 1020:GOTO"FORTH"
1020 " ." We're in the FORTH!" CR
1030 " (More Forth Commands)
1040 "BASIC" BEEP 1
1050 PRINT'We're back in BASIC!'
```

If the variable I is assigned a line number and GOTO"FORTH" is performed, the Forth source code is executed from the specified line number. The Forth word BASIC switches back to BASIC operation in the same line.

## 4.3 String variable

String variables are realized in Forth with the following definitions.

```
400": 1-(n-n-1)
405" 1 - ; (abbreviation)
410"
415": SRCH(a-an )
420" ( determines the length n a character string,)
425" (which is stored from address a and)
430" (ending in 0)
435" DUP
440" BEGIN
445" DUP C@ SWAP 1 + SWAP
450" 0= UNTIL
455" SWAP - 1- ;
460"
470": STRING(n- )
475" (Defines and dimensions string variable)
480" (of length n. When calling the string variable)
```

```
485" (puts address a and length n on the stack)
490" <BUILDS
495" ABS 255 MIN 1 MAX    (max length 255, min 1)
500" DUP C,               (enter length byte)
505" 0 DO 32 C, LOOP      (Fill with SPACES)
510" 0 C,                 (add 0 as end character)
515"DOES>
520" 1 + DUP SRCH ;       (stack starting address and length)
525"
530"0 VARIABLE IB 80 ALLOT (buffer memory of 80 characters for)
535"                      (the string input)
540": ("") ( - - a n)
545" (runtime function puts beginning address and length of the)
550" (string stored in the program on the stack)
555" R COUNT              (stack starting address and length)
560" DUP 1 + R> + >R ;    (skip saved string)
570"; " ^( -- a n)
575" (applied in the form - string" to strings)
580" ( to be entered. )
585" 94                   (ASCII value of -)
590" STATE @              (In definition?)
595" IF
600" COMPILE (^)          (compile runtime function)
605" WORD                 (read up to ^ and)
610" HERE C@ 1 + ALLOT    (save in program code)
615" ELSE                 (No definition, execute directly)
620" WORD                 (read up to ^)
625" HERE COUNT           (stack start address and length)
630" IB SWAP RED OVER     (copy to buffer IB)
635" IB SWAP 1+ CMOVE
640" 2DUP + 0 SWAP C!     (add 0 as identifier)
645" ENDIF ;
650"IMMEDIATE
655"
660": S! ( al n1 a2 n2 --)
665" (stores string at al of length n1 in string variable)
670" (at a2)
```

```
675" (Used in the form '" String" string variable S!)
680" DROP DUP 1− C@       (maximum length from variable)
685" ROT MIN 1 MAX        (no longer than max length)
690" 2DUP + 0 SWAP C!     (enter end character 0)
695" CMOVE ;              (save character string before)
```

These definitions go back to a suggestion by R. Deane published in 1980 in Dr. Dobb's "Journal of Computers” and are now almost standard.

String variables are in the form:
n STRING Variablenname

defined, where n specifies the maximum length of the character string that can be stored in it. When defining, string variables are padded with spaces. When a string variable is called, the starting address and length of the currently stored string is pushed onto the stack. Therefore, the stored character string can be output, for example, directly after the variable call using TYPE.

Strings are entered with" instead of double quotes. They can be stored in a string variable with S!:

* Text" Stringvariable S!

String operations that operate on the stacked addresses and lengths can now be defined with little effort.

Example:

```
20 STRING A$
"TEXT EXAMPLES" A$S!
A$ TYPE C
```

# Appendix A

## Error Messages

The texts of the error messages are in BASIC lines and you can change them at any time.

The errors are essentially only checked by the outer interpreter. If an error occurs when a word is called during execution, it is generally only registered after the call has ended (if at all).

| | |
|---|---|
| **"~def** | Word not defined, also not interpretable as a number<br>Example: 1234XYZ, as long as you have not defined this word. |
| **"leer** | The stack is empty Example: DEF F (warm boot) and then DROP |
| **"Zeile ~def** | BASIC line number does not exist. |
| **"~ neu** | Name already defined, the new definition has been executed and now applies in place of the old one.<br>Example: : DUP 1 PICK ; |
| **"voll** | There is not enough space left in the stack or dictionary. |
| **"nur :Def** | The word may only be used within word definitions.<br>Example: DO |
| **"~compilierbar** | The word must not be used in word definitions.<br>Example: VARIABLE |
| **"passt nicht** | Unauthorized crossing of control structures.<br>Example: IF...WHILE...LOOP |
| **"strukturfehler** | ; despite an incomplete control structure<br>Example: IF ; |

| | |
|---|---|
| **" < fence** | An attempt was made to delete a name protected by FENCE using FORGET. Example: FORGET : |
| **"~ ladend** | the word may only be entered from BASIC lines, not directly from the keyboard. Example: P> |
| **"basic** | BASIC reported a bug to Forth. Example: !: "A/3" |

When reading BASIC lines with <P , the line number is displayed with a preceding # for each error. If a negative line number is displayed, you forgot to end the import with P>.

# Appendix B

## Memory Map and cold start values

If PC-FORTH is installed, the beginning of the BASIC program memory is increased with a corresponding NEW command and PC-FORTH is loaded at the beginning of the memory (+ &C6).

The memory allocation is as follows:

| | | |
|---|---|---|
| RAM-Start | | Reserve assignment of keyboard |
| RAM-Start + 198 | ORIGIN<br>DP (HERE)<br>CSP (SP@)<br>S0 | PC-FORTH ↓<br>New definitions<br>Stack (C-Stack) ↑ |
| | R0 | Return-Stack |
| | UV | Application variables (User Variables) |
| | TOB | BASIC program memory ↓<br>BASIC variables<br>O$(0) ↑ |
| RAM-END | TIB | I$(0) |

The exact memory allocation can be determined with the MEM command from chapter 4.

The memory areas used by the Forth system are redefined with each cold start of PC-FORTH by taking the address values from specific memory addresses starting with ORIGIN. You can specify other initial values in these memory addresses and thus change the memory layout of the Forth system.

In detail, the following addresses are important for a cold start:

| | | |
|---|---|---|
| ORIGIN | + 0 jump instruction to the cold start routine | |
| | + 4 jump command to the warm start routine | |
| | + 6 jump command to hot start routine | |
| | + 8 implementation attribute &01 01 00 OC | |
| | + 12 Top word in FORTH vocabulary | |
| | + 14 UV Start of user variable range | |
| | + 16 Start of C stack | SO |
| | + 18 Start of R stack | R0 |
| | + 20 input buffer (address of I$(0) | TIB |
| | + 22 output buffer (address of O$(0) | TOB |
| | + 24 Maximum name length ( = 31) | WIDTH |
| | + 26 switch for error messages (= 0) | WARNING |
| | + 28 line number of the first message | OFFSET |
| | + 30 Top word protected against FORGET | FENCE |
| | + 32 Top word in dictionary | DP (HERE) |
| | + 34 vocabulary concatenation | VOC-LINK |
| | + 36 CPU name as a double number | |

To the left of the values are the user variables that are initialized with the values entered for these addresses during a cold start.

By changing the corresponding cold start value, for example, the two stacks can be moved to a different address range.

By changing ORIGIN + 12, ORIGIN + 30 and ORIGIN + 32 accordingly, the cold start FORTH system can be expanded to include new user definitions. These then no longer need to be compiled after a cold start.

If the FORTH system from ORIGIN to HERE is saved to a cassette with CSAVEM, the newly recorded words are available again immediately after loading.

The BASIC program memory start can be moved up without further changes to the FORTH system. Only the variables I$(0) and O$(0) must always be at the addresses TIB and TOB. Therefore, they should always be dimensioned after CLEAR or NEW first.

With the KERN command, the FORTH vocabulary can be shortened by many commands. These erasable commands are supplied as source code on the supplied cassette under the name PC-FORTH.SRC and can therefore also be changed by the user.

Since more than 3K is required in the BASIC memory for this, the memory allocation may have to be changed with NEW and specification of new cold start values for UV, SO, RO if less than 3K are available for the BASIC.

# Appendix C

# Dictionary, compiler, interpreter

## Source Code and Pseudocode

The compiler has the task of translating the Forth "source code" as entered by the user into an internal intermediate code ("pseudocode") and storing this in memory.

The source code consists of a sequence of Forth words in text form, separated by spaces, and entered by the user via the keyboard.

The internal intermediate code (pseudocode) consists of a sequence of numerical values that the compiler stores in memory. Instead of each Forth word, the compiler stores an address value as code. Each code word therefore takes up 2 bytes (16 bits).

The stored address value points to the so-called code field of the definition belonging to the entered name. This address value is therefore also called code field address (CFA).

## Inner and outer interpreter

The so-called outer interpreter is responsible for interpreting and executing the Forth words entered from the keyboard. For example, it turns on the compiler as soon as a colon is entered.

It searches the dictionary for the names entered. Once it finds a name, it calls the compiled program stored by name.

The inner interpreter is responsible for executing programs that exist as pseudocode. It fetches the pseudo code stored there from the current program address and calls up the machine program required for execution.

The outer interpreter and the compiler are both relatively slow because they have to search the entire dictionary for each command.

The inner interpreter, on the other hand, works very quickly because it gets the right addresses straight away and doesn't have to search first.

(A subprogram call in machine language takes 30 clock cycles or 23 us, the inner interpreter needs 67 clock cycles or 52 us to call the machine program belonging to the instruction. Therefore, the Forth programs compiled into the pseudo code often work almost as fast as some compilers, which translate to machine code and make heavy use of subprograms (Conversely, the faster compilers often inflate program length enormously, so the Forth code is a good compromise.)

## Dictionary and Compiler

The name entry of a Forth word is structured as follows:

| LG | Name | LK | CD | Parameter |
|---|---|---|---|---|
| t |  | T | T | t |
| NFA |  | LFA | CFA | PFA |

The most important part of the name record is the CD code field. It contains the address of the machine code program that the inner interpreter is to call. The address of the code field is called code field address CFA.

The translated (compiled) Forth program consists (almost exclusively) of consecutive code field addresses.

From the parameter field (from parameter field address PFA) the machine code program can read the parameters belonging to the relevant name while it is running.

If, for example, several constants are defined, the code fields of their name entries always contain the same address, namely the start address of the machine program "stack constant"; the value of the relevant constant is a parameter in the parameter field of the respective name entry. The "stack constant" machine program only needs this single parameter.

The name entries are linked to one another via the link field LK

When a Forth word is to be translated from source code form to intermediate code, the compiler examines the dictionary from back to front (i.e., from the most recent to the oldest name entries); it finds in the link field of each name the start address of the next oldest name entry (link field of t oldest name entry contains 0).

The name and its length are stored in the name field (from NFA). In addition to the length of the name (in bytes), the length byte LG contains further information (immediate bit and invalidity bit); the top bit of LG is always set so that when reading the name backwards, its beginning can be recognized. (This is required, for example, for the trace, which must determine the name from the code field address).

Usually, the compiler translates a printable forth word into a code field address. However, there are special cases where not one word of the source code exactly corresponds to a code field address.

Some examples:

**a)** An entered number is automatically preceded by the code field address of LIT. This measure ensures that the compiler can store the number itself in binary form in the generated program (after the LIT call). The LIT machine program simply copies the double-byte following the LIT call onto the stack and no longer needs to decode the entered number. It only has to ensure that the interpreter skips the double byte.

**b)** Words that control the compiler are executed immediately during compilation. The compiler recognizes these words because the "immediate bit" is set in the length byte of their name entry. The user can extend and modify the compiler; in order to ensure that his own compiler-extended words are not translated by the compiler but executed immediately, he must set the immediate bit in its name entry; the function IMMEDIATE is available for this purpose.

**c)** Forth words ending with the character" e.g.." generally cause the compiler to place the following information in the generated program:

— The code address of the associated runtime function whose name is given by bracketing the function name, e.g. ("")
— The character string preceded by the length byte

If the compiled function is called later, the runtime function — e.g. (.") — processes the character string following its call and causes the interpreter to continue processing after the string.

**d)** "Structure words" such as BEGIN, UNTIL etc. are basically processed as follows:

If the compiler finds the word BEGIN, it doesn't put anything in the generated program; however, it pushes the current address HERE (that is, the address from which the next code field address is to be stored) onto the stack.

The address remains there until the compiler encounters the associated word UNTIL in the course of its further work. At this point, the compiler places the code field address OBRANCH in the generated program and then the relative return address, which is calculated (with the help of the BACK function) from the top stack entry and the current storage address.

If the generated function is called later, the OBRANCH function causes the interpreter to conditionally return according to the relative reverse jump address stored after the OBRANCH.

The Forth compiler gets by with the jump functions BRANCH and OBRANCH; all jumps generated from structure words are implemented using these two functions.

Structure checks are performed during compilation.

Certain Forth words may only be specified in groups; e.g. the user must not use the word LOOP without first typing DO. Violations of this rule (as well as improper nesting, etc.) are commonly referred to as "structural errors". The following procedure allows the compiler to detect structural errors:

**a)** At the beginning of each word definition (i.e. when the compiler processes the colon), the current contents of the stack pointer are saved in the user variable CSP ("current stack position"). At the end (that is, at the semicolon), the stack pointer must have the same value again; otherwise the error message "Structure error" is returned and the invalid bit in the name entry is not cleared.

**b)** The first word of each phrase causes the compiler to push a tag (1, 2, 3, or 4) for the opened phrase onto the stack. If a word follows later that may only appear in an opened word group, the compiler checks whether the correct word group is currently opened. (If not, the error message "does not fit") appears. If the compiler finds a word that marks the end of a group of words (like UNTIL or END), then (if the word matches) the stack entry is removed again.

This procedure (also common with other compilers) has the advantage of eliminating all recognizable errors at compile time, so that the generated program is no longer burdened by time-consuming structural error queries later (when it is running).

The following indicators are used:

DO .. LOOP and DO .. + LOOP use the number 3 as identifier
Use BEGIN .. AGAIN and BEGIN .. UNTIL 1
Use IF .. ENDIF and IF .. ELSE .. ENDIF 2
CASE: .. ;use CASE 4

BEGIN .. WHILE .. REPEAT work a bit more complicated:

BEGIN initially stacks 1,

WHILE has the same effect as IF , but increases the flag set by IF from 2 to 4 so that the group of words cannot be terminated with ENDIF;

REPEAT produces an unconditional return like AGAIN (after briefly saving its own return address and current tag 4 and examining and eliminating the underlying tag 1 stacked by BEGIN), then decrements tag 4 by 2 again (because the CFA in the definition line of REPEAT acts like 2 — ) and completes its work just like ENDIF (also checking that the current tag just decremented is now 2).

Annotation:

The word group CASE: .. ;CASE can use the same identifier (namely 4) that is used in the meantime in BEGIN .. WHILE .. REPEAT, because if the two word groups were nested incorrectly, one of the structure error checks would respond in any case.

# Appendix D

# MC-12 commands

To simplify access to the MC-12 system from FORTH, a number of additional functions have been incorporated into the PC-FORTH system.

In general, of course, it is possible to use all MC-12 BASIC commands within PC-FORTH, since BASIC subprograms can be called by FORTH and parameters can be passed. So the MC-12 user can easily define FORTH words for MC-12 functions that call short BASIC subprograms (see Chapter 4).

Of course, this method is unsuitable for time-critical functions. For this reason, PC-FORTH has been expanded to include functions that enable quick access to the hardware elements and the measured value memory of the MC-12.

In order to avoid superfluous conversions, all measured values (including those stored in buffers) are treated as unsigned binary numbers, as supplied by the hardware of the MC-12 system — regardless of whether the MC-12 system is in the bipolar or unipolar mode operating mode.

At the same time, however, conversion routines were made available which convert measured values specifying the measuring range (RANGE) into a signed integer which corresponds to the measured value in mV units. These functions allow the FORTH programmer to convert exactly where necessary.

8-bit measured values are in the range between 0 and 255. 0 corresponds to the minimum in the respective measuring range, 255 to the maximum. In the case of bipolar measurement, the zero point is therefore at 128. When measuring with 11 bits, the values are between 0 and 65536 (more precisely &FFE0), the higher-order byte corresponds exactly to the 8-bit value. The zero point of a bipolar measurement is therefore at 32768 (&8000).

The values can be converted to millivolts at any time using the conversion functions 'MV' and 'DMV'.

Most of the following FORTH words have the same function as the corresponding MC-12 BASIC commands. You can also refer to the MC-12 manual for the effect of each command. In the following description, the commands are arranged according to subject groups.

## Accessing the MC-12 hardware

| | |
|---|---|
| **MCON** | ( - )<br>Turns on the MC-12 system, which has a built in time delay that allows the MC-12 hardware to stabilize. |
| **MCOFF** | ( -- )<br>Turns off the MC-12 system. |
| **SWITCHON** | ( a - )<br>Turn on the switch numbered a. The numbers 1 to 4 identify the CMOS switches 1 to 4 of the MC-12, the numbers 5 and 6 the relays RMTO and RMT1 of the CE-150. |
| **SWITCHOFF** | ( a - )<br>Turn off switch number a. The meaning of the numbers is the same as for SWITCHON. |
| **OUTCHA** | ( an- )<br>Sets the value n at analog output a. n is between 0 and 255, a is the number of the output channel. |
| **SETRANGE** | ( an- )<br>Sets the measuring range n at input a. The input channels of the MC-12 are numbered from 1 to 5, the measurement ranges from 1 to 11. Range 1 is the least sensitive range and 11 is the most sensitive. |
| **RANGE** | ( a - n )<br>Returns the number n of the measuring range currently set on channel number a. |

## Access to the measured value memory

| | | |
|---|---|---|
| **BUFNUM** | ( - n ) | Returns the number of buffers. |
| **BUFLEN** | ( - n ) | Returns the length of the buffer memory. |
| **?DBUF** | ( -f ) | Returns a 1 if the buffers contain 16-bit values (initialized with DBUFINIT), otherwise a 0. |
| **BUFOPEN** | ( an- ) | Deletes the buffer a and assigns it to the measuring range N. |
| **BUFRANGE** | ( a - n ) | Returns the measurement range of values stored in buffer a. |
| **BUFREAD** | ( a1 a2 - n ) | Returns the value n stored in buffer a1 at position a2, depending on ?DBUF in the format of 8- or 11-bit measured values. |
| **BUFWRITE** | ( a1 a2 n - ) | Writes the value n to position a2 in buffer a1, which can be an 8-bit or 16-bit value depending on the ?DBUF. |

## Other tools

| | | |
|---|---|---|
| **2@:** | (-d ) | will be in the form:<br>2@: "basicexpression"<br>used (see "@:") and returns the value of the BASIC expression as a double number. |